A Code Motion Framework for Global Instruction Scheduling*

Rajiv Gupta

Dept. of Computer Science, Univ. of Pittsburgh, Pittsburgh, PA 15260, USA

Abstract. A framework for global instruction scheduling is developed that progressively improves a given instruction schedule by eliminating delay slots in the schedule one at a time through the application of global code motions. The elimination of a delay slot is carried out in two steps: a goal oriented search which identifies a global code motion or a cascade of code motions that eliminate the delay without introducing additional delay slots in critical areas of the program; and a transformation step in which the code motion is performed along with compensation code placement and application of code optimizations enabled by code motion. The framework is powerful in that it incorporates code hoisting, including speculative hoisting, code sinking, and integrates partial redundancy elimination and partial dead code elimination optimizations with the code motion transformations. The framework is flexible in that it allows formulation of new as well as existing powerful scheduling strategies, allows restrictions to be placed on the type of instructions involved in code motion, the program regions from which the instructions can be selected for code motion, and the extent to which code motion alternatives are explored. The framework is efficient because it computes the required data dependence information on demand from the program control flow graph. Thus, it neither requires exhaustive precomputation of all data dependencies and nor their representation in form of a dependence graph prior to scheduling.

1 Introduction

Instruction scheduling and code optimizations are two important tasks performed by the compilers for superscalar and VLIW architectures. Not only are the two tasks individually complex, interactions among code optimizations and instruction scheduling exist that further affect the overall performance of the generated code. A variety of instruction scheduling techniques have been developed [3, 7, 8, 11]. These techniques differ in the program representations they use, the types of transformations they are able to perform, and the heuristics used to determine the order in which program regions are scheduled. Often there is no clear separation between the power of these techniques (i.e., the transformations they can perform) and the heuristics they use to carry out the selection of the

* Supported in part by HP Labs, Intel Corp., the NSF PYI Award CCR-9157371, and NSF Grant CCR-9402226 to the Univ. of Pittsburgh.

transformations. Data dependencies in the program must be computed and represented in a suitable form before instruction scheduling can begin. Generally it is assumed that the code optimization phase precedes the scheduling phase and no optimizations are performed during instruction scheduling. However, transformations that are performed during instruction scheduling may enable additional optimizations. Since the program representations used during code optimization typically differ from those on which instruction scheduling is performed, interleaving or integration of instruction scheduling and global code optimizations is not feasible.

In this paper we present a code motion framework that addresses the above issues. The task of scheduling is viewed as that of progressively improving upon an existing instruction schedule till no further improvements are possible. An initial instruction schedule is first generated by using local basic block scheduling. Delay slots in the current schedule are identified and removed one at a time using global code motion. The removal of a delay slot is carried out in two steps. The first step searches for a suitable code motion transformation that can eliminate the given delay slot and the second step carries out the transformation. The first step embodies the scheduling heuristics while the second step implements an integrated code motion transformation and optimization algorithm. The distinctive features of this framework are as follows:

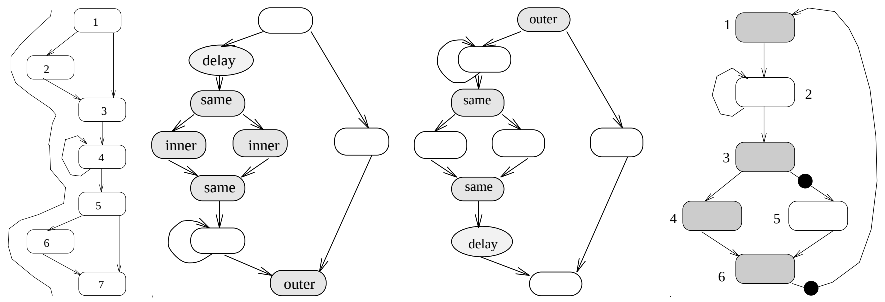
- A *goal-oriented search* algorithm for finding a suitable code motion transformation is provided. This algorithm can be adapted to explore code motions of varying complexity for the elimination of a given delay slot depending upon its criticality. The flexibility in exploring code motions allows existing scheduling strategies as well as new powerful ones to be formulated.
- A powerful set of code motion transformations is supported. It is possible to directly identify a sequence of code motions called *cascading* code motions that eliminate the delay slot without introducing additional delay slots at other critical points in the program. The identified code motions may involve code hoisting or code sinking. Furthermore, the framework is capable of performing *speculative code motions* along frequently executed paths identified through profiling.
- The transformation algorithm is an *integrated* algorithm that effects a desired code motion transformation identified by the search algorithm. It performs placement of compensation code, exploits opportunities for partial redundancy elimination (PRE) enabled by code hoisting, and exploits opportunities for partial dead code elimination (PDE) enabled by code sinking.
- Each of the above steps is formulated as a data flow problem on the control flow graph. Data dependence information is computed on *demand* as an instruction for code motion is being searched for and the code motions are performed directly on the control flow graph representation of the program. This overcomes the problem of the expensive nature of existing techniques that precompute data flow information exhaustively.

In the next section we provide an overview of the framework and in later sections we present more details.

2 Overview of the Code Motion Framework

The overall approach to scheduling is to first carry out local basic block scheduling and then identify delay slots present in the schedule. One by one these delay slots are targeted for removal through global code motion. In this section we highlight the key characteristics of the scheduling framework. Additional details are presented in the subsequent sections.

Scheduling Regions: Given a delay slot, a scheduling region is constructed to define the scope over which code motion is performed to eliminate the delay slot. The scheduling region is a multiple entry multiple exit subgraph of the control flow graph that contains the delay slot. Since we only allow acyclic scheduling, only nodes from the loop enclosing the delay slot may be included in this subgraph. While nested loops may appear in the scheduling region, they do not play a significant role in the scheduling process. In particular, code motion between different loops is not permitted. The paths in the region are divided into two categories, *critical* paths and *non-critical* paths, based upon path profiling information [1]. For the purpose of constructing these paths, each nested loop is viewed as being replaced by a single summary node. The elimination of the delay slot is carried out through a code motion that can only improve the schedules along critical paths. Thus if speculative code motion is performed it must not adversely affect any critical path while it may adversely effect non-critical paths. A subset of statements along the critical paths that are to be considered as candidates for code motion are also identified.



(a) (b) (c)

Fig. 1. (a) Critical Paths. (b) Allowed Targets for Hoisting and Sinking. (c) Scheduling a Critical Path.

The above approach of identifying a set of critical paths is more flexible than one in which a set of critical nodes alone are identified. For example, in the flow graph shown in Figure 1(a) there are four paths. Path profiles may indicate that the path corresponding to the true evaluations of both conditionals is far more frequently executed than all other paths. This information cannot be expressed by identifying critical nodes alone. Since all nodes are covered by the critical path, the consequence of specifying all nodes as critical implies that all four paths are critical.

Allowed Code Motions: Once a delay slot has been selected for removal, target areas in the program are identified from which a statement will be hoisted or sunk to fill in the delay slot. The global code motion framework presented in this paper provides a great deal of flexibility in selecting targets. Code can be moved among control equivalent regions as well as control dependent regions. The allowed code motions are summarized in Figure 1(b). In case of hoisting we can select statements from same level, inner or outer level and speculative hoisting is allowed. In case of sinking we can select statements that are at outer or same control levels. Sinking from inner level is not allowed because we do not support predicated execution. Statements can be moved across loops but not between a loop and an outer nesting level. This is because we assume that loop invariant code motion precedes code scheduling (i.e., no opportunities for moving code out of loops exist) and the movement of code from outside a loop to inside a loop is not generally beneficial.

Scheduling through Delay Elimination: The task of scheduling is viewed as sequencing code statements along program paths such that the stalls or delays expected during their execution are minimized. First a simple local basic block scheduling is performed and then all delay slots remaining in the code are targeted for elimination through global code motion. Global code motion progressively improves the schedule till no more delays that can be removed using code motion can be eliminated. The order in which the delays are removed is determined using profiling information. A delay with a high execution count is eliminated before another delay with a lower execution count. As program areas with high execution count are satisfactorily scheduled, their code orderings are marked as final to prevent future code motions from introducing compensation code in these areas. In some situations a critical path in the program may not contain any delay. However, it still may be possible to speedup this path by propagating instructions from it to other less frequently executed paths [11]. This task can be performed by artificially introducing delays in the less frequently executed parts to serve as destinations for instructions that may be moved from the more critical paths. Consider the example in Figure 1(c). It shows a scheduling region which contains the critical path 1-2-3-4-6. Since node 2 represents a loop, this node in the path does not participate in code motion. However, code statements in nodes 1, 3, 4 and 6 can be reordered to eliminate delays in along the critical path. If no delays are present in the region one can be introduced at the least important of the exits from the critical path (either edge 3-5 or edge 6-1). Code from the critical path can then be sunk to this delay to reduce the length of the critical path.

Cascading Code Motions: Often when an instruction is moved to fill a delay slot, a new slot may be created at the point where the instruction originally resided. Scheduling algorithms that focus on an immediate program region, ignore delays that may be introduced in other regions as a result of code motion. They essentially hope that either future code motions will be successful in eliminating these delays or that these delays will not significantly effect program performance. The global code motion algorithm that is presented in this paper

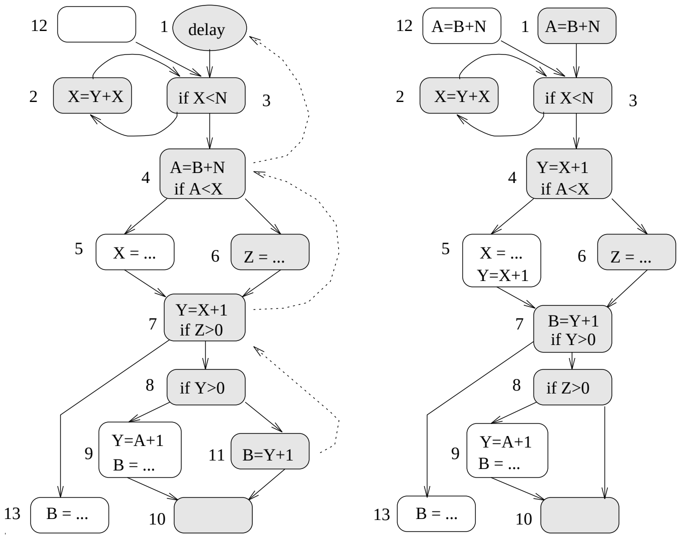


Fig. 2. Cascading and Speculative Code Motions.

is capable of determining a sequence of code motions, cascading code motions, such that the delay is either truly eliminated or shifted to a non-critical part of the program.

The example in Figure 2 illustrates cascading code motions for a given scheduling region. In order to eliminate the delay shown at node 1, instruction A=B+N is moved from node 4 to node 1. This results in the creation of another delay following the branch in node 3 and prior to the branch in node 4. This delay is filled by moving instruction Y=X+1 from node 7 to node 4. Again a delay is created between the statement in node 6 that defines Z and the conditional branch in node 7 that uses Z. This delay is filled by moving instruction B=Y+1 from node 11 to node 7. Thus, in order to remove the original delay slot in node 1, a sequence of three code hoisting motions are performed.

Speculative Code Motions: The global code motion framework described in this paper also incorporates speculative code hoisting. In the example of Figure 2 the last two code motions in the cascade, that is, the motion of Y=X+1 from node 7 to 4 and of B=Y+1 from node 11 to 7, are speculative in nature. The characteristic of a speculative code motion is that although it benefits the paths along which the hoisting is carried out, it may actually increase the execution time along other paths. In the example of Figure 2 while the path 1-3-4-6-7-8-11-10 benefits from speculative code motion, the paths that go through nodes 5 and 9 have additional instructions introduced along them.

To guide speculative code motion, we rely on the scheduling regions that have been identified using profiling data. During code motion, speculation is only applied if it benefits a critical path or a set of critical paths without having detrimental affect on any critical path. Thus, speculation only introduces instructions along paths that are considered to be non-critical.

Resource and Data Dependence Constraints on Code Motion: An instruction being moved to a delay slot may need to satisfy certain resource constraints if it is to eliminate the delay slot. For example if at the position of the delay slot the floating point addition unit is busy, then moving an instruction that requires this unit will not eliminate the delay slot. Resource constraints are computed at the outset (i.e., before the search for an instruction to fill the delay slot begins) by examining the code surrounding the delay slot under consideration.

An instruction satisfying the resource constraints can be moved into the delay slot only if it satisfies certain data dependence constraints. In particular, for a code motion to be legal the instruction may need to satisfy constraints on the variables that it is permitted to define and use. These constraints are computed on demand as the search for an instruction to fill the delay slot progresses. The advantages of demand-driven analyzers have been demonstrated in [2].

Data dependences along critical paths are maintained separately from non-critical paths. In Figure 2 the data dependence information for the path 3-4-6-7-8-11-10 is maintained separately from all other paths. The consequence of this approach is that at node 7 we determine that along critical paths a code statement containing a use of X can be hoisted to node 4 while along other non-critical paths such a statement cannot be hoisted. This makes it possible for us to discover the motion of $Y = X + 1$ from node 7 to node 4.

Search Strategies: Global code motion strategies, and thus global instruction scheduling strategies, of varying complexity and scope can be easily formulated in this framework. By limiting the scope of the scheduling region one can control the scope of the search for an instruction to fill a delay slot. Cascading motions are only performed within the scheduling region. The search for an instruction terminates once the targeted delay is eliminated. Thus, following code motion the targeted delay slot is eliminated and either no other delay slot is introduced or if one is introduced it is at a non-critical point in the program. It is also possible that the search for a sequence of code motions ends unsuccessfully. However, this will only happen if either no instruction is found to eliminate the targeted delay slot or a cascade of code motions is found which leads to the shifting of the delay slot from the original position to some other point along some critical path. At this point a number of alternatives are possible. If the delay is smaller in the number of cycles or it is in a less frequently executed node than the original delay, we may go ahead with the code motion. Another approach is to backtrack and search for alternate cascades.

3 Code Motion Search Algorithms

In this section we present algorithms that perform forward/backward search for cascade of hoistings/sinkings that eliminate a given delay slot. We first derive the conditions under which a given statement s can be moved in place of a delay slot d. Let the set of paths in the *scheduling region* of a delay slot d be denoted as P and the *critical* and *non-critical* sets of paths by $\mathcal{P}_c$ and $\mathcal{P}_{nc}$. The set of statements that are *candidates* for code motion are denoted by $\mathcal{C}$.

Let us consider the potential *hoisting* of statement s in $\mathcal{C}$ to fill the delay slot d. We divide the paths that are affected by the motion of s to d into three distinct sets: (a) $P_c(d \rightarrow s) \subseteq \mathcal{P}_c$ is the set of **direct critical** paths that contain a subpath from from d to s; (b) $P_{nc}(d \rightarrow s) \subseteq \mathcal{P}_{nc}$ is the set of **direct non-critical** paths that contain a subpath from d to s; and (c) $P(\overline{s}) \subseteq \mathcal{P}$ is the set of **crossing** paths that do not go through s but go through a node that lies along a path from d to s. The conditions that must hold for motion of s to d to be legal and beneficial for speculative and non-speculative code motion are described next.

Conditions for Speculative Hoisting of s to d: Speculative hoisting is performed if it is legal and it benefits critical paths but is not harmful to any critical path. For this to hold, the following conditions must be true:

(a) **Anticipability along direct critical paths**: For the code motion to be beneficial to the paths in $P_c(d \rightarrow s)$, the motion of s should not be blocked by any data dependence along these paths, that is, s is *anticipable* along each path in $P_c(d \rightarrow s)$.

(b) **No critical crossing paths**: The speculative motion of s will be harmful to the paths in $P_{nc}(d \rightarrow s)$ and $P(\overline{s})$. Therefore these paths must be non-critical, that is, all crossing paths in $P(\overline{s})$ must also be non-critical.

(c) **Liveness along direct non-critical and crossing paths**: If s defines variable v, then the speculative motion of s may block any previously reachable definitions of v to some points along the non-critical paths in $P_{nc}(d \rightarrow s) \cup P(\overline{s})$. Therefore for the speculative code motion to be legal, we must make sure that along portions of above non-critical paths to which previously reachable definitions of v are now blocked, v is *not live*. In particular for any given direct or crossing non-critical path p through a node n, such that s is *not anticipable* along p at n, v must not be live at n along p.

Note that if predicated execution is supported, the conditions for predication based hoisting can be obtained by modifying the above rules for speculative hoisting. The modification simply ignores the constraints placed by the liveness conditions above.

Conditions for Non-speculative Hoisting of s to d: A non-speculative hoisting is performed if it is legal and benefits all paths that it effects. For this to hold, the following conditions must be true:

(a) **Anticipability along direct paths**: The code motion should be beneficial to all paths from d to s. Thus, s must be *anticipable* along all paths in $P_c(d \rightarrow s)$ and $P_{nc}(d \rightarrow s)$.

(b) **No crossing Paths**: There should be no crossing paths, that is, the set $P(\overline{s})$ should be empty.

Conditions for Sinking of s to d: Since we do not support predicated execution in this framework, the sinking of s to d can be performed only in certain restricted situations. For the sinking to be legal the following condition must be true:

(a) **Availability along all paths**: The statement s should be *available* at d along all paths leading to d. Thus s must dominate d and along each path from s to d no data dependences should block the sinking of s to d.

While it is possible to derive conditions for predication based sinking in a manner analogous to speculative hoisting, additional issues must be addressed for allowing such sinking. In particular, the framework must support mechanisms for deriving the predicate and assessing whether the benefit of eliminating the delay outweighs the cost of computing the predicate.

The final condition that a code motion must satisfy for it to be useful is that it should indeed eliminate the delay slot. This condition is stated below.

Conditions for delay elimination: A code motion that moves a statement s to a delay slot d eliminates the delay slot if the following conditions are true:

(a) **Resource availability:** The resources required for execution of s are free at d; and
(b) **Data availability:** The values of all operands used by s are expected to be available at d.

Next let us consider the problem of searching for a cascade of code motions that eliminates a given delay slot. We consider cascades constructed through hoisting. Similar algorithm can be developed for cascades based upon sinking. A simple greedy algorithm can be used to construct the cascade. We can examine the candidate statements in $\mathcal{C}$ that are reachable from the delay slot d one by one and check them for the code hoisting and delay elimination conditions. The statements in $\mathcal{C}$ that are closer to the delay slot are examined before the ones that are further away. When a statement s that can be moved to d is found, it is added to the cascade. If hoisting of s creates a new delay at s's original position, the above process is repeated to eliminate the newly created delay. In this manner the cascade is built till finally a motion that does not create a delay is found and the construction of the cascade is complete, or a delay is encountered that cannot be removed. In this case we can backtrack and explore other alternatives. Another approach for searching would be to examine all candidates in $\mathcal{C}$ reachable from d before selecting one for hoisting. In this case we can give preference to the statement that does not create a new delay. Thus, this approach is likely to result in shorter cascades than the greedy approach.

Example. Next we show how the above approach can discover the cascade of code motions shown in Figure 2. The scheduling region shown contains one critical path from delay at 1 to the exit at node 10 which is 1-23-4-6-7-8-11-10. Here node 23 represents a single summary node representation of the loop nested in the scheduling region. We also assume that all assignments along the critical path are candidates for code motion. Of all the other paths originating at node 1, paths ending with the subpath 9-10 or 7-13 is a non-critical path. Now let us consider each code motion in the cascade and verify that they satisfy the required conditions presented earlier in this section:

$A = B + N@4 \rightarrow delay@1$: Since statement $A = B + N$ is anticipable at 1 along all paths that contain a subpath from 1 to 4, this code motion constitutes a legal non-speculative hoisting.

$Y = X + 1@7 \rightarrow delay@4$: Statement $Y = X + 1$ is anticipable at 4 along the direct critical path 1-23-4-6-7-8-11-10. Furthermore Y is not live at 4 along the direct non-critical paths containing the subpath 4-5-7 along which $Y = X + 1$ is not anticipable at 4. There are no crossing paths in this case. Therefore this code motion represents a legal speculative hoisting.

$B = Y + 1@11 \rightarrow delay@7$: Statement $B = Y + 1$ is anticipable at 7 along the direct critical path 1-23-4-6-7-8-11-10. Furthermore B is not live along the crossing paths that end with the subpath 7-13 or 7-8-9. Moreover these crossing paths are non-critical. Finally along all direct non-critical paths which contain the subpath 7-8-11, $B = Y + 1$ is anticipable. Thus this code motion also represents a legal speculative hoisting.

The above cascade will be discovered by the greedy search algorithm since in each of the above cases the first candidate along the critical path is successfully hoisted.

We now discuss the details of the data flow analysis required to check the conditions for code hoisting/sinking stated earlier. The data flow analysis is performed in a *demand-driven* fashion in that we only perform analysis relevant to the condition being evaluated. Since the same data flow information may be needed to check conditions for motion of multiple statements, the results of analysis are cached to avoid their recomputation. Another characteristic of this analysis is that it is *path-sensitive*. The anticipability and liveness conditions require that the data flow information be computed with respect to different categories of paths. Conventional analysis techniques compute data flow with respect to all program paths. An approach for path-sensitive analysis was introduced in [4, 5] where in addition to computing a data flow fact at a program point, the set of paths along which the fact holds is also computed.

The *anticipability* analysis is performed by starting at node s and propagating data flow information backward along nodes that lie along paths from d to s (set of these nodes is denoted by $N(d \rightarrow S)$). In the data flow equations given below $Apaths_s(n)$ evaluates to the subset of paths in $P_c(d \rightarrow s) \cup P_{nc}(d \rightarrow s)$, denoted as $P(d \rightarrow s)$, along which s is anticipable at n. In the equations $OnPaths(n)$ denotes the subset of paths in $P(d \rightarrow s)$ that pass through n. The predicate $Pres_s(n)$ is in the equations is 1 if n does not block the hoisting of s and 0 otherwise.

$$\mathbf{Apaths_s(n)} = \begin{cases} \phi & \textbf{if } \mathbf{Pres_s(n) = 0} \\ \mathbf{P(d \rightarrow s)} & \textbf{elseif } \mathbf{s \in n} \\ \mathbf{OnPaths(n)} \cap \bigcup_{\mathbf{m \in Succ(n)}} \mathbf{Apaths_s(m)} & \textbf{otherwise} \end{cases}$$

The *liveness* analysis is performed by starting at nodes reachable from d and propagating data flow information backward along nodes that lie along paths that pass through d. Let us assume that the variable defined by s is v and therefore we are interested in liveness information for v. In the data flow equations given below $Lpaths_v(n)$ evaluates to the subset of paths in $\mathcal{P}$ that pass through d along which v is live at n. In the equations $OnPaths(n)$ denotes the subset of paths through d that also pass through n and the sets $Def(n)/Use(n)$ contain variables defined/used by n.

$$\mathbf{Lpaths_V(n)} = \begin{cases} \mathbf{OnPaths(n)} & \mathbf{if\ v \in Use(n)} \\ \phi & \mathbf{elseif\ v \in Def(n)} \\ \mathbf{OnPaths(n)} \cap \bigcup\limits_{\mathbf{m \in Succ(n)}} \mathbf{Lpaths_V(m)} & \mathbf{otherwise} \end{cases}$$

In the above analyses for computing sets of paths along which anticipability or liveness is true, a set of paths can be represented by a bitvector in which each bit corresponds to a unique path. Thus, an intersection/union operation of sets of paths can be performed by bitwise and/or operations.

Code hoisting requires path-sensitive liveness and anticipability analysis. The sinking of a statement to a delay requires availability analysis. In our framework it is only possible to sink a statement to a delay slot if it is available at the delay slot along all paths leading to the delay slot. Thus, the availability analysis is not path-sensitive and is same as conventional availability analysis. In the data flow equations below $Avl_s(n)$ evaluates to 1 if s is available at n along all paths leading to n and to 0 otherwise. The analysis is perform over set of nodes along paths from s to d. This set of nodes is denoted by $N(s \rightarrow d)$. The predicate $Pres_s(n)$ is 1 if n does not block the sinking of s and 0 otherwise. It should be noted that if predication based sinking were allowed, path-sensitive analysis would also be required for code sinking.

$$\mathbf{Avl_S(n)} = \begin{cases} \mathbf{1} & \mathbf{if\ s \in n} \\ \mathbf{0} & \mathbf{elseif\ Pred(n) - N(s \rightarrow d) \neq} \phi \\ \mathbf{Pres_S(n)} \wedge \bigwedge\limits_{\mathbf{m \in Pred(n)}} \mathbf{Avl_S(m)} & \mathbf{otherwise} \end{cases}$$

In the above analyses the cost of data flow analysis is controlled by demand-driven computation of data flow information for subgraphs relevant to the code motion under consideration. For example, anticipability information is only computed over subgraphs for only those statements that are considered for hoisting and liveness information is computed over subgraphs for only those variables that are defined by statements considered for hoisting. In contrast conventional exhaustive analysis would compute anticipability of all statements and liveness of all variables at all program points.

The anticipability and availability analysis for a statement essentially discovers the absence and presence of dependences along program subpaths involving the variables defined/used by the statement. Since other statements may define/use the same variables, this information can be reused when they are considered for code motion along the same paths. Similarly the same liveness information may also used repeatedly. Thus, the data flow information must be cached for reuse. The form of the cache that is maintained for the three types of data flow information that we compute is as follows:

$AntiCache(p, n_i \rightarrow n_j) = \{(DEF, \overline{DEF}), (USE, \overline{USE})\}$: For the path $p \in \mathcal{P}$, if a statement s, that defines a variable from DEF and only uses variables from USE, is anticipable at n_j's entry then it is also anticipable at n_j's entry along p. On the other hand if the statement either defines a variable from $\overline{DEF}$ or uses a variable from $\overline{USE}$, it is not anticipable at n_i's entry.

$LiveCache(p, n_i \rightarrow n_j) = (LIVE, \overline{LIVE})$: The variables in $LIVE(\overline{LIVE})$ are live(not live) at all nodes along the subpath of p from n_i to n_j.

$AvlCache(n_i \rightarrow n_j) = \{(DEF, \overline{DEF}), (USE, \overline{USE})\}$: If a statement that defines a variable from DEF and only uses variables from USE is available along all paths at n_i, then it is also available along all paths at each node that lies along a path from n_i to n_j (including n_j). On the other hand if the statement either defines a variable from $\overline{DEF}$ or uses a variable from $\overline{USE}$, it is not available along paths at n_j's entry.

In the above cache the DEF, USE and $LIVE$ sets indicate the absence of dependences involving certain variables while the sets $\overline{DEF}$, $\overline{USE}$ and $\overline{LIVE}$ indicate the presence of dependences. It should be noted that the absence of dependences is maintained for the longest possible subpaths along which the condition holds while the presence of dependences is maintained for the shortest possible subpaths along which the condition holds.

Let us see how the entries in $AntiCache$ are filled in following anticipability analysis. For each path p in $Apaths_s(d)$ the absence of dependences between s and statements along the subpath from d to s is detected by the analysis. Therefore the variable defined by s is added to $AntiCache(p, d \rightarrow s).DEF$ and the variables used by s are added to $AntiCache(p, d \rightarrow s).USE$. If during the analysis s is blocked by node n (i.e., $Pres_s(n) = 0$), then a dependence between s and n has been detected. If s is anti or output dependent upon n along path p then variable defined by s is added to $AntiCache(p, n \rightarrow s).\overline{DEF}$. If s is flow dependent upon n along path p then variable defined by n is added to $AntiCache(p, d \rightarrow s).\overline{USE}$. The caches for liveness and availability results are similarly handled.

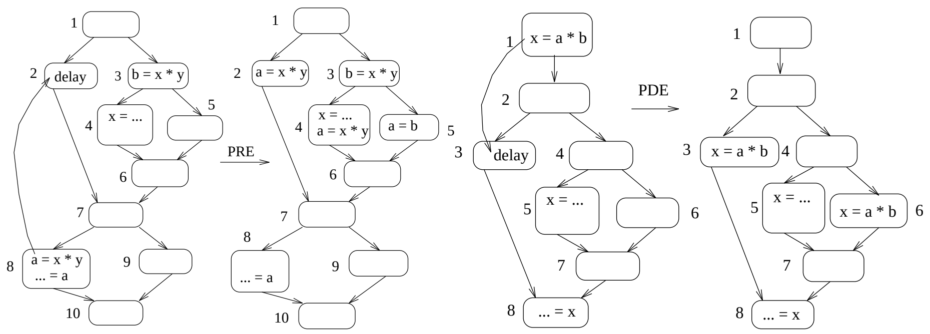
The contents of $AnitCache$ are used as follows. Consider a subpath of a critical path p which is $d \rightarrow \cdots \rightarrow A = B + 1 \rightarrow X = B + C$. Let us assume that anticipability analysis determines that $A = B + 1$ cannot be hoisted to d along p due to a definition of B along the way at node n. In this case B will be added to the set $AntiCache(p, n \rightarrow A = B + 1).\overline{USE}$. Thus if later an attempt is made to hoist $X = B + C$ to d, by consulting the cache, and thus without any further analysis, we can determine that $X = B + C$ cannot be hoisted to d since it also uses B. Similarly in other situations we may be able to determine that a statement can be hoisted to a certain program point by consulting the cache.

The cost of analysis over the subgraphs can be further reduced using demand-driven analysis techniques presented in [2]. These techniques answer a data flow query at a program point by searching for its solution starting at the program point. The search is terminated when the solution is known. Thus, we can often expect the search to terminate before examining the entire subgraph. The above approach requires reversal of the analysis which can be readily performed for distributive problems such as anticipability, liveness, and availability problems.

4 Code Motion Transformation Algorithms

Next we discuss the generation of compensation code associated with the code motion. The algorithm that determines the placement of compensation code during hoisting also performs PRE and the algorithm for generating compensation code during sinking performs the PDE optimization.

The PRE optimization eliminates redundant evaluations of expressions that may be present along certain program paths. An expression evaluation is considered to be partially redundant if along some program path(s) leading to the evaluation, the expression has already been evaluated while along other path(s) no prior evaluation exists. Opportunities for PRE arise if the instruction scheduler performs code hoisting to generate a fast schedule. However, traditional approach for generating compensation code is unaware of such opportunities.



(a) Partial Redundancy Elimination. (b) Partial Dead Code Elimination.

Fig. 3. Optimizations Enabled by Code Motion.

Consider the first example flow graph of Figure 3(a). Assume that the path 1-2-7-8-10 through the acyclic flow graph is the most frequently executed and therefore is the object of code reordering being performed by the scheduler. Along this path the evaluation of expression $x * y$, that is, the value of a, immediately precedes the use of a. Given that the multiply operation is a long latency operation, the scheduler may choose to hoist the evaluation of $x * y$ to node 2 assuming the hoisting is not blocked by other statements. Notice that in doing so the expression is being speculatively hoisted above node 7. Following code reorganization, compensation code is introduced. A straightforward approach will introduce the compensation code at node 6. However, as shown in the second flow graph in Figure 3(a), by placing compensation code instead at nodes 4 and 5 an evaluation of $x * y$ along path 3-5-6 can be avoided.

We have developed an algorithm for compensation code placement which performs PRE. This algorithm is an extension of the PRE algorithm proposed by Knoop et al. [9]. The first extension is to handle speculative code motion. The second extension enables discovery of a placement in which one of the positions is the delay slot. More details can be found in [6]

The PDE optimization eliminates dead executions of statements that may be present along certain program paths. An execution of an assignment statement is considered to be partially dead if along some program path leading from the assignment, the value computed by the assignment is never used. Opportunities for PDE arise if the instruction scheduler performs code sinking to generate a good schedule. Consider the first flow graph of Figure 3(b). Assume that the path 1-2-3-8 through the acyclic flow graph is the most frequently executed and therefore is the object of code reordering. Along this path the scheduler may choose to sink the assignment $x = a * b$ to node 3 to eliminate the delay that may be present at node 3. Following code sinking compensation code is introduced. A straightforward approach will introduce the compensation code at node 4. However, as shown in the second flow graph in Figure 3(b), by placing compensation code instead at nodes 5 and 6 an execution of assignment along path 1-2-4-5-7-8 can be avoided.

In the above example the statement being sunk was live along the critical path being scheduled. It is possible that the partially dead statement being sunk is dead along the critical path. In such situations acyclic schedulers sink the statement to the point where the statement becomes fully dead and hence can be entirely eliminated from the critical path.

The algorithm for compensation code placement is a straightforward adaptation of the PDE algorithm proposed by Knoop et al. [10]. The adaptation essentially enforces the restriction on the placement of scheduled instruction much in the same manner as it was done for the PRE algorithm.

5 Adapting the Framework

Next we illustrate the flexibility and power of our framework by comparing it with other scheduling techniques and demonstrating that these techniques can be emulated by simply restricting the power of the presented framework. Primarily by limiting the global scope over which code motion can be performed is sufficient to emulate these techniques.

Trace Scheduling [3]: In this technique the critical part is limited to a trace which is a sequence of consecutive basic blocks along a program path. A trace cannot extend across loop boundaries. The data dependence graph for the trace is constructed and a scheduler (e.g., a list scheduler) is used to generate the instruction schedule. The resulting schedule may contain delay slots that are left in the schedule. This technique is equivalent to performing cascading motions within the trace to minimize delay slots. In contrast in the presented framework multiple critical paths can be selected that extend across loop boundaries, it is able to push the delay slots out of the critical paths to less important points of the program, it does not require the construction of an explicit data dependence graph, and instead of scheduling all of the instructions from scratch its code reordering efforts are aimed at eliminating identifiable delay slots. Trace scheduling does not exploit PRE/PDE opportunities enabled by code motion.

Region Scheduling [7]: Here first a program dependence graph representation of the program is constructed and then instruction scheduling is carried out by first reordering code within a control region and then performing code motions across control dependent regions. This technique like the presented framework also progressively improves the instruction schedule through code motions. However, speculative code motion can only be performed in very limited situations. For the example of Figure 2, the speculative motion of Y=X+1 from node 7 to node 4 cannot be performed during region scheduling. In addition, the space and time overhead of constructing the program dependence graph is significant. Other PDG based scheduling techniques also suffer from similar drawbacks and do not incorporate PRE and PDE optimizations.

Superblock Scheduling [8]: In this technique before instruction scheduling is carried out, tail duplication is performed to eliminate code following merge points in the program. Code reordering performed on the resulting superblocks does not result in any compensation code. However, this simplification comes at the cost of significant code growth. The framework presented in this paper supports a powerful algorithm for generating optimized compensation code and therefore avoids unnecessary code growth due to tail duplication.

Critical Path Reduction [11]: In this technique the scheduling region being scheduled consists of a multiple entry multiple exit acyclic region. The region exits are classified into two categories, frequently taken and infrequently taken. Even if the paths leading to frequently taken exits does not include any delay slots, an attempt is made to reduce the schedule length by pushing statements off these paths and to the infrequently taken exits. This is sometimes possible because a path to a frequently taken exit may contain statements that are dead along these paths even though they may be live along other paths. The presented framework can emulate critical path reduction as follows. We can assume that delays are present at infrequently taken exits and initiate code sinking search to fill these delay slots by sinking of code statements from the critical paths. Only the statements that are dead along the most frequently taken exit may be considered as the candidates for code sinking. Since our compensation code generation algorithm already incorporates the PDE optimization, after code sinking no compensation code would be introduced along paths leading to frequently taken exits if the moved statement is dead along those paths.

6 Conclusion

This paper presented an approach for global instruction scheduling that views the task of instruction scheduling as an application of series of code motion transformations. A powerful search algorithm for finding code motions that can eliminate a targeted delay slot is developed. Once the desired code motions have been identified, a code motion algorithm is provided to perform the code motion, introduce compensation code required to preserve program semantics and apply PRE and PDE optimizations that may be enabled by the code motion. The framework is not only powerful but also quite versatile, as it can be easily adapted to emulate existing scheduling strategies as well as formulate new ones. We believe that this has been made possible by a clear separation of a scheduling step into two phases: the search phase that looks for opportunities for code motion and the transformation phase that actually performs the transformation. The search algorithm and the transformation algorithm compute the necessary data flow upon demand and thus costly computation of all data dependencies and their representation in form of a dependence graph is avoided.

References

1. T. Ball and J. Larus, "Efficient Path Profiling," *29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, 1996.
2. E. Duesterwald, R. Gupta, and M.L. Soffa, "A Practical Framework for Demand Driven Interprocedural Data Flow Analysis," *to appear ACM Transactions on Programming Languages and Systems*, November 1998.
3. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, 30(7), July 1981.
4. R. Gupta, D.A. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pages 102-115, San Francisco, California, November 1997.
5. R. Gupta, D.A. Berson, and J.Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," *30th IEEE/ACM Intl. Symp. on Microarchitecture*, pages 558-568, Research Triangle Park, NC, Dec. 1997.
6. R. Gupta, "Code Optimization as a Side Effect of Instruction Scheduling," *Intl. Conf. on High Perf. Computing*, pages 370-377, Bangalore, India, Dec. 1997.
7. R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Trans. on Software Eng.*, 16(4):421-431, 1990.
8. W-M. 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*, A:229-248, 1993.
9. J. Knoop, O. Ruthing, and B. Steffen, "Lazy Code Motion," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 224-234, 1992.
10. J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 147-158, 1994.
11. M. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Programs," *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1995.

This article was processed using the LaTeX macro package with LLNCS style