Code Optimization as a Side Effect of Instruction Scheduling *

Rajiv Gupta†
Department of Computer Science
University of Pittsburgh
Pittsburgh PA 15260
gupta@cs.pitt.edu

Abstract

An instruction scheduler utilizes code reordering techniques for generating schedules in which instructions can be issued without delays. In order to perform code reordering across branches, code motion is performed that hoists some instructions above branches and sinks others below branches. Following code reordering, compensation code must be introduced in order to preserve program semantics. In this paper we demonstrate that several important code optimizations can be performed as a side effect of generating compensation code. These optimizations include partial redundancy elimination, partial dead code elimination, elimination of redundant loads, and elimination of dead stores. We demonstrate how existing data flow frameworks for these optimizations can be extended for generating optimized compensation code.

Keywords - *code hoisting, code sinking, code optimization, compensation code, redundancy elimination, dead code removal.*

1 Introduction

The task of an instruction scheduler for a superscalar or VLIW architecture is to generate code orderings along program paths such that instructions can be continuously issued without pipeline delays. Towards this end typically long latency instructions, such as floating point instructions and memory operations (loads and stores), are separated from those instructions that cannot execute before the completion of long latency instructions. In order to generate desirable instruction schedules, instructions may have to placed at earlier points in the program (i.e., hoisted) or placed at later points in the program (i.e., sunk). The movement of instructions across branches requires introduction of compensation code along certain program paths to preserve the program's semantics.

Code optimizations that result in redundancy elimination and deadness removal also perform hoisting and sinking to achieve optimization. However, the placement of code statements as determined by the optimizer need to be consistent with the placement of the same statements as determined by the scheduler. Typically, this problem is ignored by first performing code optimization without any regard to instruction scheduling and later in a separate phase performing instruction scheduling. In this approach the scheduler may undo many of the optimizations that were performed by the optimizer. Meanwhile new optimization opportunities may become available due to code reordering performed by the instruction scheduler [3]. These opportunities go unexploited.

In this paper an approach that integrates code optimizations into an instruction scheduler is presented. Code optimizations that involve code motion are not performed prior to instruction scheduling. Instead they are incorporated into the task of compensation code generation. The instruction scheduler is free to order the instructions in a manner that generates fast schedules. The instructions that are involved in code motion during this process are passed on to compensation code placement algorithms. These algorithms identify placements that satisfy the following characteristics: (a) they preserve program semantics; (b) they are consistent with the instruction schedule; and

†Supported in part by grants from Hewlett Packard, Intel Corp., and NSF Presidential Young Investigator Award CCR-9157371 to the Univ. of Pittsburgh.

(c) they generate optimized compensation code.

Since maximum benefit can be derived by optimizing long latency operations, we assume that the instructions involving such operations are the target of optimization during compensation code generation. Compensation code for other statements can be generated using traditional approaches [8, 9]. The optimizations that we present include: (a) partial redundancy elimination [13, 11] of hoisted expressions involving scalar variables and floating point operations during acyclic scheduling; (b) partial dead code elimination [12, 7] of sunk assignments involving scalar variables and floating point operations during acyclic scheduling; (c) elimination of partially redundant loads during cyclic scheduling; and (d) elimination of partially dead stores during cyclic scheduling. In the subsequent sections the integration of above optimizations and compensation code generation is illustrated and the details of optimization algorithms are discussed.

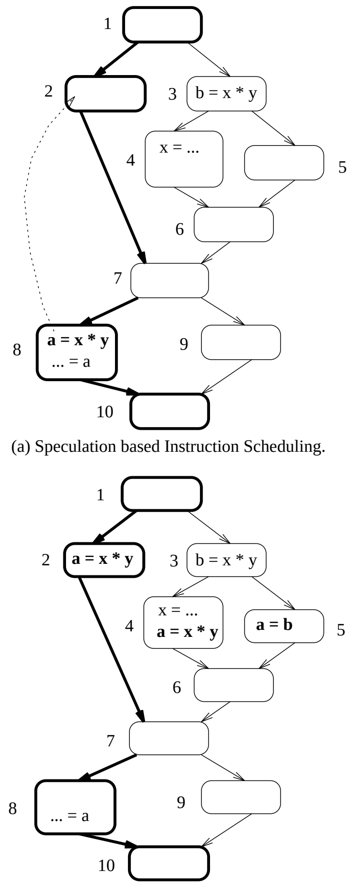
2 Acyclic Scheduling

We first consider the optimizations of partial redundancy elimination involving expression evaluations and partial dead code elimination involving assignments where the expressions and statements involve references to scalar variables. In our discussion we assume that instruction scheduling is being performed along acyclic paths using a trace scheduler [8] although the techniques are applicable to other types of acyclic schedulers. Furthermore, we assume that the scheduler is capable of performing code motion that relies on speculation and predication features supported by modern processor architectures.

2.1 Partial Redundancy Elimination

This 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. One approach to this optimization is to hoist a partially redundant expression evaluation to a set of program points such that at some of these points the expression is fully redundant and hence its evaluation can be removed.

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. Consider the example flow graph of Figure 1a. 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 trace scheduler [8]. 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 Figure 1b, by placing compensation code instead at nodes 4 and 5 an evaluation of $x * y$ along path 3-5-6 can be avoided. This example illustrates that PRE opportunities can be exploited by appropriate placement of compensation code.



(b) Compensation Code Placement.

Figure 1: Partial Redundancy Elimination.

Next we develop an algorithm for compensation code placement which performs PRE. This algorithm is an extension of the PRE algorithm proposed by Knoop et al. [11]. The key points that must be addressed by the compensation code placement algorithm are as follows:

1. Speculation based Scheduling: The scheduler may perform speculative code motion as illustrated by the example in Figure 1. Knoop's algorithm cannot perform speculative code motion. Thus, to discover the solution of Figure 1b, we must extend Knoop's algorithm to allow speculative code motion. However, this does not imply that the compensation code placement can arbitrarily apply speculation since the application of speculation is typically limited to situations where it is determined to be useful. Thus, the compensation code placement algorithm should only perform the speculation that was applied by the scheduler, that is, speculation that was determined to be profitable by the scheduler.

2. Restrictions on Code Placement: Knoop's algorithm hoists the code as far as it is necessary to exploit all PRE opportunities that can be exploited using traditional code hoisting (i.e., code hoisting without speculation). In our scenario the scheduler has already determined the placement of an expression evaluation along the frequently executed path. Thus, we are looking for a specific hoisting in which the restriction on the placement of one copy of the expression is imposed by the scheduler while the placement of compensation code copies can be carried out to exploit PRE.

Our algorithm consists of three steps which are described next. In the description of these steps we will point out how the above extensions to Knoop's algorithm are achieved. For simplicity, the algorithm's discussion considers the generation of compensation code for a specific expression e that has been hoisted to the target node $target_e$ by the scheduler. The analysis is performed over the original code, that is, the code prior to reordering by the scheduler. In a practical implementation, all hoisted expressions can be considered simultaneously.

1. Down-Safety Analysis: In this step all program points to which expression e can be hoisted safely are determined. This is carried out through backward propagation of e from all points at which it is generated. The expression is allowed to propagate backwards as long as the data dependences do not block its propagation. When the expression reaches the node $target_e$, its propagation along that path is discontinued. When a split (conditional) node n is encountered, where n is not the node $target_e$, then the expression is allowed to propagate to node n's entry if one of the following conditions hold: (a) e is computed in n and anticipatable at n's entry; (b) e is preserved by n and it has been propagated to the entries of each of n's successors; and (c) e is preserved by n, speculation of e was performed by the scheduler at n, and e has been propagated to the entry of at least one of n's successors. This condition has been included to allow speculation that is used by the scheduler. The combined effect of this condition and the discontinuation of propagation above the node $target_e$ ensures that the placement of e found by the PRE algorithm will include as one of its points, the node $target_e$.

The equations for computing down-safety are given below. With each conditional node n in the acyclic graph a variable, $Spec_e(n)$, is associated which is set to 1 if the acyclic scheduler performs hoisting of e above n using speculation; otherwise it is set to 0. $Used_e(n)$ is 1 for nodes that computed expression e and 0 otherwise. $Pres_e(n)$ is 1 for nodes that preserve e and 0 for all other nodes. The solution to the following equations, i.e., one bit variables $Dsafe_e(n)$, is interpreted as follows. If $Dsafe_e(n)$ is 1 then it is possible to propagate e to n's entry; otherwise e cannot be propagated to n's entry.

$$Dsafe_e(n) = \begin{cases} if\ n = exit \vee n = target_e\ then \\ \quad 0 \\ else \\ \quad\quad Used_e(n)\ \vee \\ (Pres_e(n) \wedge \bigwedge_{m \in Succ(n)} Dsafe_e(m))\ \vee \\ (Spec_e(n) \wedge \bigvee_{m \in Succ(n)} Dsafe_e(m)) \end{cases}$$

To perform analysis for a single expression we have used single bit variables. If multiple expressions are to be considered, then a bit vector can be performed in which each bit corresponds to a unique expression.

2. Earliestness Analysis: This forward analysis phase is identical to the one proposed by Knoop et al. [11]. It identifies the earliest code placement points which are down-safe. The down-safety analysis discussed above guarantees that the $target_e$ node will be one of the earliest points. All remaining points are points at which compensation code is placed.

3. Delayability Analysis: In this final step additional analysis is performed to find the latest points at which expressions can be placed without increasing redundancy. This approach leads to the reduction in the size of the lifetime of the computed expressions by the compensation code. This step is similar to the implementation described by Knoop et al. in [11] except for one simple modification. The placement of the expression by the scheduler at the node $target_e$ cannot be delayed.

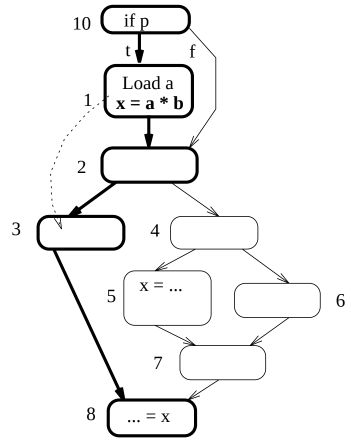
Let us revisit the example in Figure 1. During down-safety analysis the expression $x * y$ will be allowed to propagate above node 7 since speculation at this node is enabled. The earliest point at which compensating copy of $x * y$ is to be placed is found to be node 4. This also happens to be the latest point. The original occurrence of the expression is removed from node 8 and copies are placed in nodes 2 and 4. At node 5, where the expression is already available, a simple copy assignment is introduced.

2.2 Partial Dead Code Elimination

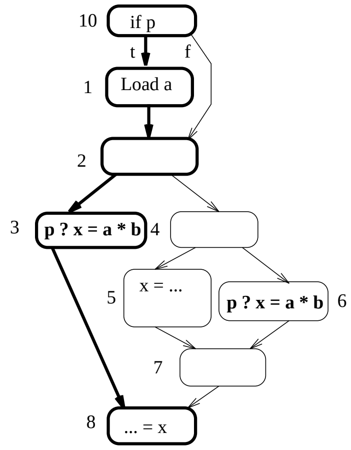
This optimization eliminates dead executions of statements that may be present along certain program paths. An execution of an assignment statement is considered to be partially dead if along some program path leading from the assignment, the value computed by the assignment is never used. One approach to this optimization is to sink the partially dead assignment to a set of program points such that at some of these points the statement is fully dead and hence its execution can be removed [12].

Opportunities for PDE arise if the instruction scheduler performs code sinking to generate a good schedule. Consider the example flow graph of Figure 2a. Assume that the path 10-1-2-3-8 through the acyclic flow graph is the most frequently executed and therefore is the object of code reordering. Along this path the execution of assignment $x = a * b$ is immediately preceded by the loading of a. Given that the load is long latency operation, the scheduler may choose to sink the assignment to node 3. Notice that in doing so predication is being used to sink the statement past node 2. Following code sinking compensation code is introduced. A straightforward approach will introduce the compensation code at node 4. However, as shown in Figure 2b, by placing compensation code instead at nodes 5 and 6 an execution of assignment along path 10-1-2-4-5-7-8 can be avoided.

In the above example the statement being sunk was live along the trace being scheduled. It is possible that the partially dead statement being sunk is dead along the trace. 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 trace [14]. For example, if the use of x from node 8 is removed, the partially dead assignment



(a) Predication based Instruction Scheduling



(b) Compensation Code Placement.

Figure 2: Partial Dead Code Elimination.

$x = a * b$ is dead along the trace 10-1-2-3-8. In this case if the scheduler sinks the assignment to node 3 as before, it can be entirely eliminated. Notice that in this situation also the opportunity for PDE by eliminating an execution of the assignment along the path 10-1-2-4-5-7-8 can be exploited during compensation code generation. The above examples illustrate that PDE opportunities can be exploited by appropriate placement of compensation code.

Next we develop an algorithm for compensation code placement which performs PDE. This algorithm is an extension of the PDE algorithm proposed by Knoop et al. [12]. The key points that must be addressed by the compensation code placement algorithm are as follows:

1. Predication based Scheduling: The scheduler may perform predication enabled code sinking as illustrated by the example in Figure 2. Knoop's algorithm cannot perform such code motion. Thus, to discover the solution of Figure 2b, we must extend Knoop's algorithm to allow predication enabled code sinking. However, this does not imply that the compensation code placement algorithm can arbitrarily apply predication since the application of predication is limited to situations where it is determined to be useful by the scheduler. Thus, the compensation code placement algorithm should only perform predication that was actually performed by the scheduler.

2. Restrictions on Code Placement: Knoop's algorithm sinks an assignment as far as necessary to exploit all PDE opportunities that can be exploited using traditional code sinking (i.e., code sinking without predication). In our scenario the scheduler has already determined the placement of an assignment execution along the frequently executed path. Thus, we are looking for a specific sinking in which the restriction on the placement of one copy of the assignment is imposed by the scheduler while the placement of compensation code copies can be carried out to exploit PDE.

The steps of algorithm are described next. In the description of these steps we will point out how the above extensions to Knoop's PDE algorithm are achieved. For simplicity, the algorithm's discussion considers the generation of compensation code for a specific assignment statement s that has been sunk to the target node $target_s$ by the scheduler. The analysis is performed over the original code, that is, the code prior to reordering by the scheduler. In a practical implementation, all assignments that are involved in sinking can be considered simultaneously.

1. Assignment Delayability Analysis: In this step all program points to which sinking of s is allowed is determined. This is carried out through forward propagation of s from its original position. The statement is allowed to propagate forward as long as the data dependences do not block its propagation. When the statement reaches the target node $target_s$, its propagation along that path is discontinued. When a merge node n is encountered, where n is not the $target_s$ node, then the statement is allowed to propagate to node n's exit if one of the following conditions hold: (a) s is computed in n and available at n's exit; (b) s is not blocked by n and it has been propagated to the exits of each of n's predecessors; and (c) s is not blocked by n, predication of s was performed by the scheduler at n, and s has been propagated to the exit of at least one of n's predecessors. This condition has been included to allow predication enabled sinking that is used by the scheduler. The combined effect of this condition and the discontinuation of propagation past the node $target_s$ ensures that the placement of s found by the PDE algorithm will include as one of its point the node $target_s$.

The equations for computing delayability are given below. With each merge node n in the acyclic graph a variable, $Predication_s(n)$, is associated which is set to 1 if the acyclic scheduler performs sinking s below n using predication; otherwise it is set to 0. $Compute_s(n)$ is 1 for the node containing s and 0 for all other nodes. $Block_s(n)$ is 1 for nodes that block s's propagation and 0 for all other nodes. The solution to the following equations, i.e., one bit variables $Delay_s(n)$, is interpreted as follows. If $Delay_s(n)$ is 1 then it is possible to propagate s to n's exit; otherwise s cannot be propagated to s's exit.

$$Delay_s(n) = \begin{cases} if\ n = start \vee n = target_s\ then \\ \quad 0 \\ else \\ \quad\quad Compute_s(n)\ \vee \\ (\neg Block_s(n) \wedge \bigwedge_{m \in Pred(n)} Delay_s(m))\ \vee \\ (Predication_e(n) \wedge \bigvee_{m \in Pred(n)} Delay_s(m)) \end{cases}$$

To perform analysis for a single assignment we have used single bit variables. If multiple assignments are to be considered, then a bit vector can be used in which each bit corresponds to a unique assignment.

2. Insertion Point Computation: This step is identical to the one proposed by Knoop et al. [12]. It identifies the code placement points for the assignment. As mentioned earlier, our formulation of delayability analysis is guaranteed to result in node $target_s$ being selected as an insertion point. All remaining points are points at which compensation code are placed.

3. Assignment Deletion: In this final step the dead assignments are removed. The removal of PDE is essentially achieved by removing all fully dead assignments placed at insertion points in the previous steps. Sinking of an assignment to some insertion points essentially causes them to become fully dead.

Let us revisit the example in Figure 2. During delayability analysis the assignment $x = a * b$ will be allowed to propagate below node 2 since predication at this node is enabled. The insertion points at which compensating copies of $x = a * b$ are to be placed are found to be nodes 5 and 6. The assignment introduced at node 5 is removed since it is fully dead. Thus, finally predicated copies of s are placed at nodes 3 and 6.

3 Cyclic Scheduling

In this section we consider optimization opportunities that arise due to load/store operations corresponding to *array* references. Furthermore these opportunities arise when code hoisting and sinking is being applied to achieve inter iteration code motion in loops. In particular, we consider the integration of an extended form of circular scheduling [10] with load/store elimination. We consider only non-speculative execution of loads and stores. However, our techniques can be extended to consider speculation for advanced architectures [2].

The circular scheduling technique was originally designed for loops whose bodies are formed by straightline code. Thus when inter iteration code motion was applied to generate good schedules, no compensation code was ever generated within the loop. However, this technique can be extended to allow loops whose bodies contain branches. The scheduler can isolate the most frequently executed trace in the loop body and hoist/sink instructions from this trace to preceding/succeeding loop iterations. Following code motion, compensation code must be generated within the loop body to preserve program semantics. We outline the compensation code generation algorithms in this paper that perform elimination of partially redundant loads and partially dead stores.

3.1 Partially Redundant Loads

Let us first illustrate the removal of partially redundant loads due to array references during extended circular scheduling. Consider two successive loop iterations of the loop in Figure 3a. The load performed in node 2 during the later iteration is partially redundant because the control may arrive at this node from node 5 in the preceding iteration. Next let us see how an opportunity to remove this partially redundant load may arise during compensation code generation following inter iteration load hoisting that may be applied by circular scheduling.

Let us assume that the path 2-3-4-6 is the most frequently executed path in the loop. The circular scheduler may decide to hoist the load in node 2 of a later iteration to node 4 of the preceding iteration so that the value of $A[i]$ would be available by the time the use of $A[i]$ in node 2 is encountered. A straightforward approach for compensation code generation may place a copy of the load at the end of node 9. However, a better placement is achieved by removing the load at node 5 and placing a load at node 7 as shown in Figure 3b. The result is that along every

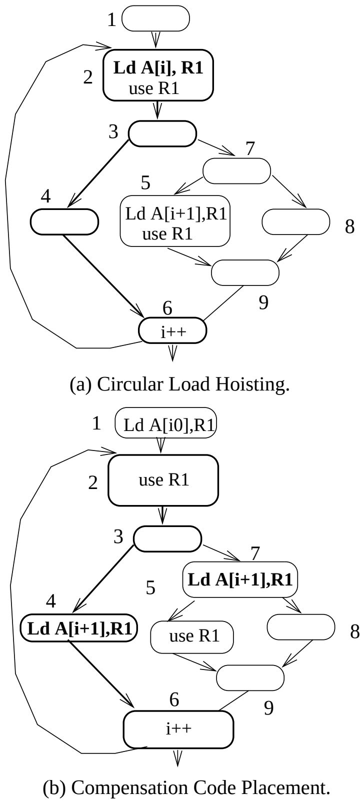
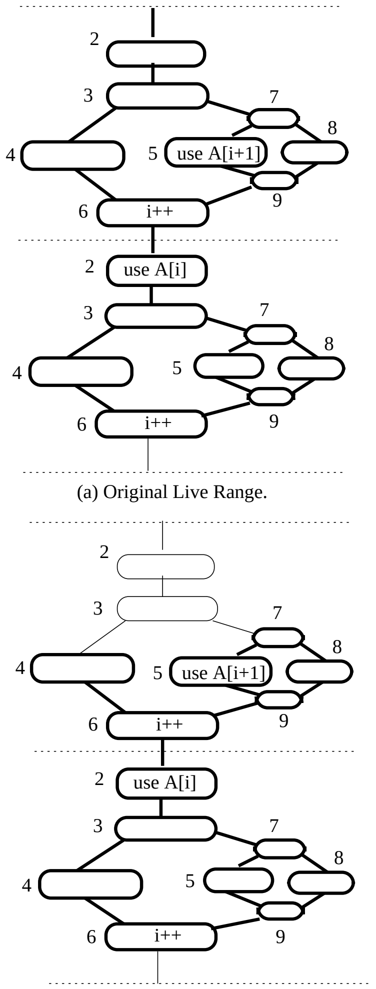


Figure 3: Eliminating Redundant Loads.

path through a loop iteration a single load is issued. On the other hand if the load was placed at node 9, along the path that passes through nodes 5 and 9 the load would be issued twice. The prologue of the resulting loop contains the load hoisted out of the first loop iteration.

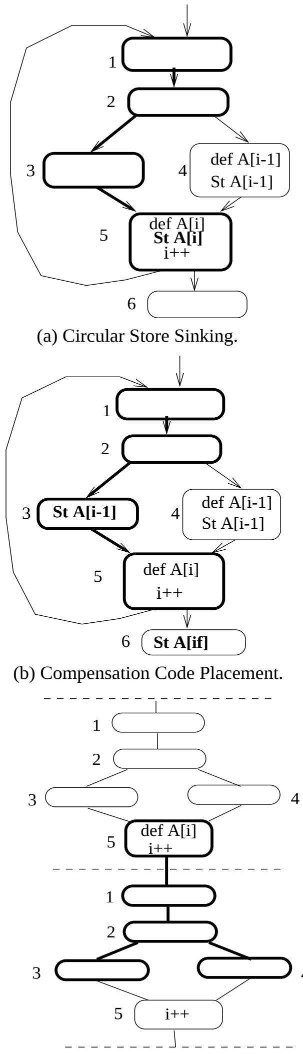
The load elimination algorithm presented in [6] can be adapted for generating optimized placement of compensation loads. The algorithm in [6] consists of the following steps. It identifies the group of array references that reference the same array element over a series of loop iterations. For example in Figure 3a references to $A[i+1]$ and $A[i]$ reference the same array element over two successive loop iterations. Next by performing *backward anticipatability* analysis and *forward availability* analysis and unioning the results of these analysis, a live range representing the value of the array element over two successive loop iterations is constructed. The resulting live range is shown in Figure 4a. Loads that are originally present are removed and loads are placed at points where the value is anticipatable and not already available, that is, at the initial points of the live range. In our example this will provide us with the single point which is node 2. However, since the scheduler has already selected a



(a) Original Live Range.

(b) Modified Live Range.

Figure 4: Live Range Construction.



(a) Circular Store Sinking.

(b) Compensation Code Placement.

(c) Live Range for Congruent References.

Figure 5: Eliminating Dead Stores.

placement for the load as node 4 we would essentially like to construct a live range which has as one of its initial points as node 4. This live range is shown in Figure 4b and it has two initial points, node 4 and node 7. Thus, the placement for the compensating load is found to be node 7 and load at node 5 is no longer present. To obtain the desired solution the backward anticipatability analysis is modified so that the propagation is discontinued when the target node 4 is encountered. As a result the resulting live range ends at nodes 4 and 7.

So far we did not consider speculative motion of loads which is supported by some architectures such as Playdoh [2]. The backward anticipatability analysis can be modified to handle speculative loads in much the same way as speculative motion of expressions was incorporated in the PRE algorithm of section 2.1.

3.2 Partially Dead Store Elimination

Let us now examine the removal of dead stores of array values during extended circular scheduling. Consider two successive loop iterations of the loop in Figure 5a. The store performed in node 5 in the earlier iteration may be performed again in the next iteration if the control arrives at node 4 in the later iteration. In other words the store in node 5 is partially dead. Next let us see how an opportunity to remove the partial deadness of this store may arise during compensation code generation following inter iteration store sinking that may be applied by circular scheduling.

Let us assume that the path 1-2-3-5 is the most frequently executed path in the loop. The circular scheduler may decide to sink the store in node 5 of an earlier iteration to node 3 of the next iteration. A straightforward approach for compensation code generation may place a copy of the store at the beginning of node 4. However, this placement is not needed since the store is dead at node 5. The optimized code is shown in Figure 3b. In this resulting code, unlike the original code, along every path through a loop iteration a single store is issued. The epilogue of the resulting loop issues the store that was sunk out of the last loop iteration.

An algorithm for store placement can be found in [6]. While the load placement algorithm requires backward anticipatability and forward availability to construct the relevant live range, the placement of stores requires forward anticipatability and backward availability analysis of definitions that refer to the same array element over a series of loop iterations. Just as loads were placed at the starting points of the live range, stores are placed at the terminating points of the live range. In order to satisfy the code placement restriction placed by the circular scheduler, the forward analysis must be restricted so that sinking of the store beyond the target node is not permitted. This will force the placement point selected by the circular scheduler to be one of the termination points of the live range. For the example in Figure 5a the resulting live range is shown in Figure 5c. The terminating points are nodes 3 and 4. Thus, the compensating store should be placed at node 4. However, since the store would be fully dead at that point, it is no longer needed.

4 Conclusion

In this paper we demonstrated the integration of several important code optimizations with compensation code generation. Instead of first generating compensation code using some simple approach and then optimizing that code, our approach directly determines optimized compensation code and its placement. Furthermore, we have shown that this approach can be used both during acyclic and cyclic scheduling. In previous work it has been shown that several DAG restructuring transformations [3] and instruction selection techniques [1] can be integrated with acyclic scheduling.

In recent work we have developed new techniques for conditional branch deletion [5] and PDE [4] that aggressively apply control flow restructuring techniques to enable optimization opportunities available along selected paths. We are examining the integration of these optimizations with code reordering along a trace.

References

[1] S. Novack and A. Nicolau, "Mutation Scheduling: A Unified Approach to Compiling for Fine-grain Parallelism," *Proc. Languages and Compilers for Parallel Computing*, LNCS 892, 1994.

[2] V. Kathail, M. Schlansker, and B. Rau, *HPL PlayDoh Architecture Specification: Version 1.0*, HPL-93-80, Feb. 1994.

[3] D. Berson, P. Chang, R. Gupta and M.L. Soffa, "Integrating Program Optimizations and Transformations with the Scheduling of Instruction Level Parallelism," *Ninth Annual Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1996.

[4] R. Bodik and R. Gupta, "Partial Dead Code Elimination using Slicing Transformations," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 575-586, Las Vegas, Nevada, June 1997.

[5] R. Bodik, R. Gupta, and M.L. Soffa, "Interprocedural Conditional Branch Elimination," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 146-158, Las Vegas, Nevada, June 1997.

[6] R. Bodik and R. Gupta, "Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures," *Intl. Journal of Parallel Programming*, 24(6):481-512, 1996.

[7] K. Ebcioglu et al., "VLIW Compilation Techniques in a Superscalar Environment," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 36-48, Orlando, Florida, June 1994.

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

[9] 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.

[10] S. Jain, "Circular Scheduling: A New technique to Perform Software Pipelining," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 219-228, 1991.

[11] J. Knoop, O. Ruthing, and B. Steffen, "Lazy Code Motion," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 224-234, 1992.

[12] J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 147-158, 1994.

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

[14] M.S. Schlansker and V.Kathail, "Critical Path Reduction for Scalar Programs," *28th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, Ann Arbor, Michigan, Nov. 1995.