INCREMENTAL COMPILATION OF LOCALLY OPTIMIZED CODE

Lori L. Pollock and *Mary Lou Soffa*

Department of Computer Science
University of Pittsburgh
Pittsburgh, Penna. 15260

Abstract - Although optimizing compilers have successfully been used to reduce the size and running times of compiled programs, present incremental compilers only support the incremental update of unoptimized code. In this work, we extend the notion of incremental compilation to include optimized code. Techniques to incrementally compile locally optimized code, given intermediate code modifications are developed using a program representation based on flow graphs and dags. A model is designed to represent both unoptimized and optimized code and to maintain an optimizing history. Changes to the optimized code which either destroy optimizations or create conditions for further optimizations are incorporated into the model and the optimized code without recompiling unaffected optimizations.

1. Introduction

The recent explosion of interactive systems has heightened interest in user friendly programming environments. A key component of these systems is an incremental compiler which is automatically invoked by source program edits in order to provide a uniform user interface. Compilation time and response time are reduced by recompiling only those statements directly changed by the programmer or indirectly affected by the change. Although substantial numbers of incremental compilation systems have been designed and implemented,[2,3,5,8,9,13,12] these systems all assume unoptimized code. In this work, we extend the notion of incremental compilation to include optimized code.

The problem of incrementally compiling optimized code is inherently difficult due to a number of factors. Code optimizing transformations complicate the mapping from the source program to intermediate code due to the deletion, replacement and reorganization of intermediate code statements. Furthermore, a change in the source program can invalidate an optimization that was done previously. In order to "undo" the optimization, it is necessary to keep a history of the effects of existing optimizations (an optimization history), realizing that the order in which the optimizations are destroyed by program changes is independent of the order in which they were performed.

Although the elimination of the effects of destroyed optimizations is sufficient to maintain the functional equivalence between source and optimized code, this can lead to continual reduction of the degree of optimization as more program changes are incrementally compiled into the optimized code. Code which was initially very optimized can become increasingly unoptimized as a result of a series of incrementally compiled program changes. Therefore, it is important to detect any newly validated optimizations and incrementally perform these additional optimizations. Another difficulty involves a rippling effect from creating and destroying optimizations. The code that is generated to reflect a destroyed or created optimization can in turn create or destroy other optimizations.

Related to the problem of incremental compilation of optimized code is the symbolic debugging of optimized code. Some programs may not be able to execute without optimization, due to time and space constraints. However, the reordering and elimination performed during optimization has impeded the use of symbolic debuggers for optimized programs. Recent work on symbolic debugging for optimized code,[7,16] using some type of optimization history, successfully demonstrates the feasibility of the symbolic debugging of optimized code. Further support for using an optimizing compiler during debugging is the increased debugging information which can be obtained for the user during the analysis performed by the optimizing phase (e.g., ud and du chains).[14,10]

Although no work has been found in the literature which directly addresses the issue of incrementally updating optimized code, some research has been devoted to incremental data flow analysis and could perhaps be applied to an incremental optimization scheme.[11,15,4]

Thus, in addition to improving the response time and recompilation time, an incremental optimizing compiler would be valuable in providing information to aid in the symbolic debugging of optimized code and thus eliminate the need for maintaining a non-optimizing compiler used for debugging and another optimizing compiler used for production. Small maintenance changes after a program is in production could be quickly incorporated without consuming computer resources for a total recompilation and recalculation of optimizations.

1.1. Features of an Incremental Optimizer

The ultimate goal of this work is the development of an incremental optimizing compiler with the requirements that the compiler:

(1) detects any change in the source program that invalidates current optimized code;

(2) detects any change in the source program that creates optimizations;

(3) correctly and efficiently incorporates changes into the optimized code and updates the optimizing history to maintain consistency between the optimized code and the optimization history;

(4) uses techniques which are consistent with, and can be integrated into, existing incremental compilers; and

(5) enables the interactive symbolic debugging of optimized code.

As a first step towards this goal, we consider local optimizations in this paper. Techniques are developed to incrementally compile code which has been optimized by multiple passes of local common subexpression and redundant store elimination. A model based on the classical flow graph and augmented dag representation is designed to maintain an optimizing history. Algorithms which detect changes that affect local optimizations and subsequently update both the model and optimized code are presented. The technique is capable of handling noncontrol statement changes as well as control flow changes which affect the basic block structure. Finally, extensions to include global, loop and peephole optimizations are discussed.

There are two approaches that can be taken in the construction of an incremental optimizing compiler. We are currently following the simpler approach which assumes that source program changes are correctly translated into a sequence of unoptimized intermediate code changes. This is a valid assumption, given the existence of incremental compilers for unoptimized code. Thus, we need only consider changes that are being made to the unoptimized intermediate code, which we assume to be three address code. With this scheme, when the intermediate code is initially generated, an optimizing phase is carried out on the entire intermediate code. Any changes are incorporated incrementally into the optimized code. The more difficult approach and one that we intend to investigate, is to have the initial optimization phase be performed incrementally as the intermediate code is generated.

1.2. Overview

In Section 2, program changes that both destroy and create conditions for local common subexpression and redundant store elimination are described as well as their highly interrelated effects. Section 3 presents the model which is used to maintain an optimization history and optimized code. Algorithms which modify the optimized code and model based on the type of intermediate code change are developed in Section 4. The algorithms are generalized in Section 5 to support the effect of flow graph changes on locally optimized code. Extensions of the technique to global and peephole optimizations are given in Section 6 followed by a section detailing work in progress and future research directions.

2. Effects of Program Change on Optimized Code

We first consider the effects of program modification on optimized code. We examine program changes that do not alter the flow graph structure and changes that create a basic block, divide a basic block into smaller blocks or combine two blocks into one. Edit changes that create, delete or replace intermediate code statements are considered as well as modifications to individual operands within an intermediate code statement. The analysis consists of examining the requirements which must be satisfied in order to correctly perform each optimization and then detailing those types of edits which either destroy or create the conditions required for the optimization.

In order that the model be capable of representing the history of all types of optimizations, the analysis of program changes was performed taking into account the requirements for global and peephole optimizations as well as local optimization. As this paper deals primarily with local optimization, we focus on program changes which could affect local common subexpression and redundant store elimination. We first examine these two optimizations in isolation and then consider their intricate interconnections.

In the following discussion, the terms "use" and "def" have the traditional meaning; that is, a "use" of an identifier A is any occurrence where the value of A is needed. A "def" of A is an assignment of a value to A. A detailed description of techniques for code optimization can be found in Aho and Ullman.[1]

2.1. Local Redundant Store Elimination

Local redundant store elimination is an optimization in which a store to variable A is redundant and may be eliminated if there exists a later store to A in the same basic block and the following 2 conditions hold: (1) Every path that reaches the later store to A performs the first store to A, and (2) there exists no use of A between the two stores. The following conditions cause a *destruction of a redundant store optimization:*

(1) Insertion of a use of A between the 2 stores.

(2) Deletion of the later store to A. It should be noted that deletion of the first store does not affect the optimized code.

(3) Changing the flow of control between the 2 stores to A, separating the stores into different basic blocks.

The following intermediate code changes cause a *creation of a redundant store elimination:*

(1) Deletion of the only use of A between 2 stores to A within a basic block.

(2) Insertion of a store to A when there exists an earlier or later store to A within the same basic block with no intervening uses of A.

(3) Changing the flow of control which results in merging 2 basic blocks such that there exist 2 stores to A within the same basic block with no intervening uses of A.

2.2. Local Common Subexpression Elimination

Local common subexpression elimination has the following requirements and actions. If there exist 2 common subexpression evaluations in the same basic block, (e.g., statement i:E=A+B... statement j:D=A+B) and the following conditions hold between the 2 subexpression evaluations:

(1) there exists no redefinition of the common subexpression operands (A or B) and (2) there exists no use or definition of the variable (D) being assigned in the later common subexpression evaluation, then the second subexpression (D=A+B) is redundant and can be replaced by a simple store (D=E) immediately after the first subexpression evaluation.

Common subexpression eliminations are *destroyed by the following:*

(1) Insertion of a store (redefinition) to one of the common subexpression operands (A or B) between the subexpression evaluations.

(2) Insertion of a store to, or use of, the variable being defined in the later expression (D) between the common subexpression evaluations.

(3) Deletion of one of the common subexpression statements completely.

(4) Changing an operand within a common subexpression evaluation.

(5) Changing the flow of control between the 2 common subexpression evaluations, separating the common subexpressions into different basic blocks.

The following are events that *create a common subexpression elimination,* assuming that the two necessary conditions for elimination hold:

(1) Changing an operand such that the resulting expression matches another expression evaluation within the same basic block.

(2) Insertion of a statement whose expression evaluation matches another expression within the same basic block.

(3) Deletion of a store (redefinition) to one of the operands (A or B) of a common subexpression.

(4) Deletion of a store to, or use of, the variable defined in the later common subexpression (D).

(5) Changing the flow of control which results in merging 2 basic blocks such that there exists local common subexpression evaluations within the same basic block.

2.3. Interconnection of Optimizations

Although each code optimization was analyzed in isolation to determine the program changes which invalidate or create conditions for each optimization, the complete effects of a particular program change can involve a combination of destruction and creation of several different optimizations. This can occur by a program change directly causing multiple optimization changes or as a result of modifying the code in order to update affected optimizations. For example, we consider the effects on the optimized code of inserting the statement Z=B+C after statement 3 in the following unoptimized code segment. The three address code has been slightly modified such that multiple operands can appear on the left hand side of an assignment. (i.e., X,B=Y+Z means X=Y+Z and B=X.)

Unoptimized code		Optimized code
1. A=Y+2		1. A=Y+2
2. Z=X+W		2. Z=X+W
3. X=Y+2	Z=B+C	3. X,B=Y+2
4. B=Y+2		

We first observe that the expression B+C is not common with any previous expression evaluations. However, the insertion of a definition of Z at this point creates a redundant store to the variable Z at statement 2. Thus, the statement Z=X+W can be eliminated in the updated optimized code. The deletion of Z=X+W from the optimized code allows further optimization of the common subexpression statements A=Y+2 and X=Y+2 since the use of X in Z=X+W is now eliminated. The use of operand B in the inserted statement between the two occurrences of the common subexpression Y+2 at statements 3 and 4 invalidates the previous common subexpression optimization, This optimization reversal requires an insertion of an evaluation of Y+2 and store to B at its original store location. The updated unoptimized and optimized code become:

Unoptimized code	Optimized code
1. A=Y+2	1. A,X=Y+2
2. Z=X+W	2. Z=B+C
3. X=Y+2	3. B=Y+2
4. Z=B+C	
5. B=Y+2	

This single intermediate code insertion causes a creation of a redundant store elimination which creates conditions for a new common subexpression elimination. The inserted statement also destroys a common subexpression optimization by inserting a use of the variable defined in the later occurrence of the common subexpression between the expression evaluations. Detection of such multiple effects of a single program change must be embedded into the incremental detection/update process for optimized code. This requires careful integration of the information obtained from the analysis of the effects of program changes.

Using the analysis as a basis for detecting conditions that affect optimizations, a model capable of maintaining the information needed to incrementally update optimized code was developed.

3. Modified Flow Graph Augmented Dag Model

The model (MFAD) is a modified flow graph with augmented dags (directed acyclic graphs) which enables incremental updates of locally optimized intermediate code. The optimization history for redundant store and common subexpression elimination can be maintained using MFAD which permits the detection of invalidated and newly validated instances of these optimizations.

The model is a variant of the traditional flow graph commonly used to portray the flow of control between basic blocks. Although following the same basic structure as the traditional flow graph, each flow graph node contains additional information including flags for various optimizations and source-to-model mapping information. Each basic block is represented by augmented dags which have traditionally been used to illustrate the code dependencies within a straight line code segment. The dag is modified such that the following labels appear on each node. The first two labels are also characteristic of the traditional computation dag.

(1) Each leaf is labeled with a unique id (variable name or constant) representing the value of the leaf on entry to the basic block.

(2) Each node is labeled by an operator symbol representing the operation which computes the value of the node.

(3) Nodes are optionally labeled by a set of variable labels. In particular, those identifiers which at some time during the execution of the basic block, have the computed value represented by that node. Each variable label is composed of three parts and takes the form:

VARIABLE NAME	STORE STATUS	ORIGINAL LOCATION	
		NODE POINTER	LABEL INDEX
(string)	(store/ nostore)	(pointer)	(integer)

The **variable name** identifies the variable associated with the value computed at the node. Used to maintain a history of eliminated variable stores, the **storestatus** may reflect either a STORE or NOSTORE value. The value STORE indicates that there currently exists a store of this node's value to the variable, while a NOSTORE implies that there is currently no store. The status is NOSTORE if the store has been eliminated by optimization.

Used to maintain a history of common subexpression optimizations, the **original location** component of the label contains adequate information to determine the precise location of the original store of this node's value to the variable. In a traditional dag construction, a common subexpression is represented by several variables labeling the operator node of the common subexpression in order to evaluate the expression once and store the computed value into each labeling variable.[1] However, the position of the original store to each labeling variable is needed during incremental optimizing compilation. This is used to determine whether the optimization is invalidated and to correctly relocate the store.

The *node pointer* points to the dag node corresponding to the original location of the store to the labeling variable prior to optimization. That is, if the original store to a variable v occurred at the unoptimized statement s mapped to node **n**, regardless of whether or not the store was moved by optimization, v's *node pointer* will point to node **n**. If optimization removes node **n** from the dag (the common subexpression evaluation at s does not exist in the optimized code), v's *node pointer* will point to the node built just prior to node **n** (i.e., the node corresponding to the unoptimized statement immediately preceding the statement s). Therefore, if no common subexpression optimization is performed to affect the variable's original store location, its *node pointer* points to the node containing the variable's label. This is illustrated in Figure 1.

The existence of nodes with more than one variable label suggests that the mapping between the unoptimized code and MFAD is a many-to-one mapping. That is, a single dag node can represent several unoptimized intermediate code statements, and in particular, those statements corresponding to stores to the variables labeling that node. Furthermore, the node corresponding to the original location of a variable store can represent several unoptimized statements (i.e., contain several variable labels). In this situation, the *node pointer* of that variable label is insufficient to precisely determine the original position of the variable's store. We know only that the original store occurred at one of the unoptimized statements mapped to node **n**. In the example below, we cannot determine by looking at the model whether the original store to P occurred after the store to D or the store to G. Additional information is required to explicitly identify which unoptimized statement mapped to node **n** corresponds to the original store to the variable. This is needed in order to keep a complete accurate history of the common subexpression optimizations so detection of affected optimizations and updates to the optimized code can be correctly performed.

Therefore, the *label index* is included as part of the variable label to specify which unoptimized statement that maps to the target node of the variable's *node pointer* represents the variable's original store position. If no common subexpression has been performed such that the location occurs at the node on which the variable label currently resides, then the *label index* is set to zero. The combined use of the *node pointer* and *label index* information suffices to uniquely determine the original location of a variable's store in the unoptimized code.

A chain of *node pointers* will exist within the dag when consecutive later occurrences of different common subexpressions occur. Figure 1 illustrates this scenario.

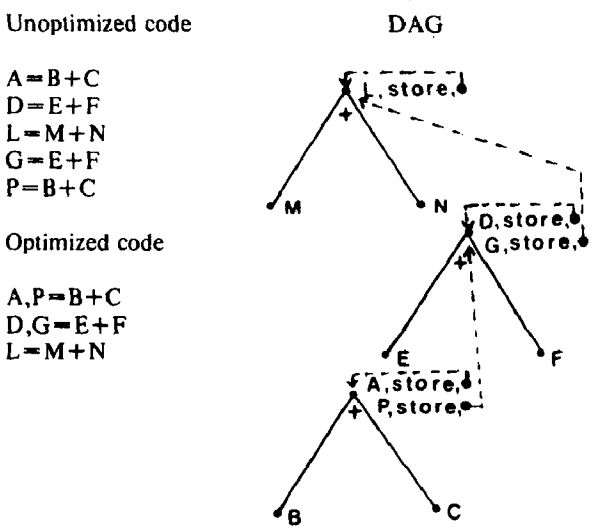

Figure 1: Node pointers in MFAD.

As the statement G=E+F is optimized and replaced by a store immediately following D=E+F (the first occurrence of this common subexpression), a variable label for G is added to the node labeled by D with its *node pointer* set to the node labeled by L representing the statement immediately preceding the original store to G. Similarly, as the statement P=B+C is optimized and replaced by a store immediately following the statement A=B+C, a variable label for P is added to the node labeled by A with its *node pointer* set to the node (labeled by G) representing the statement immediately preceding the original store to P. The *label indices* of both G and P are set accordingly. The two consecutive common subexpression optimizations create a chain of *node pointers* for the unoptimized statement P=B+C. The entire chain is needed to avoid losing the order of the unoptimized code as common subexpressions are created and destroyed.

Additional labels on nodes in MFAD include:

(4) Each interior node includes a set of pointers to nodes which reference it. These back pointers are used to ensure that all *node pointers* are kept current. For example, if variable v labeling node n is deleted, all *node pointers* pointing to node n with label indices referencing variable v should be updated to point to the node corresponding to the statement preceding the one being deleted.

(5) All root nodes which correspond to statements contain a pointer to the node's generated target instruction sequence. This provides the mapping between the model and optimized code.

Each intermediate code statement is represented by a unique subtree in one of the augmented dags of the flow graph. The nodes of the dag for a basic block are ordered by performing a postorder traversal of the nodes in the same order in which they were created. Following the flow graph node order, this postorder traversal of each dag structure yields the optimized code sequence for the entire program. The unoptimized intermediate code can be obtained by performing the same traversal with variable stores delayed until the end of their respective *node pointer* chains and variable stores inserted at each NOSTORE variable label.

Figure 2 gives an example illustrating the MFAD representation of optimizations.

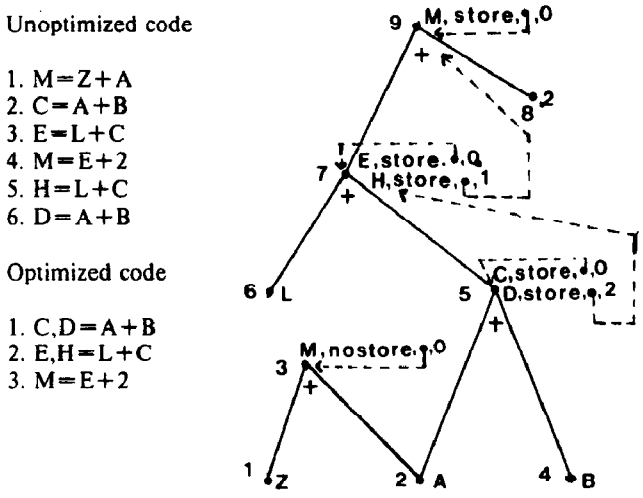

Figure 2: Optimizations Represented in MFAD.

MFAD is related to the model developed by Hennessy[7] for the symbolic debugging of optimized code. In symbolic debugging, it is necessary to detect whether a variable's value is current (i.e., correctly displayed according to the user's viewpoint) at some point in the program's execution and attempt to recover those values which are noncurrent. In order to accomplish this, Hennessy presents a model which is also based on the traditional flow graph and dag models, but requires a less extensive optimization history than MFAD. For example, Hennessy's work does not require the exact original location of a later common subexpression evaluation. The close similarity of these models supports the claim that MFAD can be used for both symbolic debugging and incremental compilation of optimized code.

MFAD is purposely designed to take advantage of the inherent qualities of the flow graph and dag models and their long standing, successful implementation in conventional optimizing compilers. The traditional optimization phase can be slightly modified to construct the initial MFAD such that the additional information which is readily available at certain points in the optimization process, is saved rather than destroyed.

4. Incremental Algorithms - Locally Optimized Code

Incremental optimization algorithms were developed which use MFAD to detect and update the effects of invalidated and newly validated optimizations caused by intermediate code changes. The algorithms were constructed by directly applying the information obtained from the analysis of the effects of program edits on the optimizations (see Section 2) to MFAD. Given that a particular action is known to cause invalidation of an optimization, the model is used to correctly detect whether execution of that action has, in fact, destroyed an instance of the optimization. If it has, the model and mappings are updated as well as the optimized code to reflect the reversal of the optimization. Similarly, when an action is known to create a necessary condition for an optimization, the algorithms use the model to determine that all conditions for the new optimization are satisfied.

4.1. Design Considerations

The analysis discussed in Section 2 demonstrates the intricate interconnections of the optimizations and the potential for a single program change to affect several optimizations. Loss of information during detection and update of one optimization can possibly cause another affected optimization to go undetected. Therefore, to ensure correct detection and avoid redundant actions, the algorithms are designed according to the type of program edit and with careful examination of the order of detection and updates.

Since intermediate code level changes can be classified as insertion, deletion, or replacement of an intermediate code statement, separate algorithms were developed for incrementally compiling each of these basic program changes. Multiple intermediate code changes are then handled by recursively applying these algorithms. An alternative approach is to design separate algorithms for each type of optimization being supported by the compiler. Although this may more clearly distinguish the detection and update processes required for each optimization, this approach was not taken because it appears to be more difficult to deal with the interrelationships of the optimizations.

4.1.1. Order of detecting affected optimizations

The order that we detect and update affected optimizations is independent of the order in which the optimizations were originally performed. However, careful ordering of detection and update in response to a single program change must be done to ensure correctness as well as improve the efficiency of the incremental optimization process.

Some ordering must be imposed on the algorithms in order to ensure that correct information is available during certain detection operations. For example, when checking for the effects of deleting a statement, the model representation of the deleted statement should be maintained until all checks are performed.

Efficiency is improved by determining early in the detection algorithm if the inserted statement is a redundant store which can be immediately eliminated. The model is updated to represent the inserted redundant store statement, but all

other checks normally performed during an insertion are avoided, since the statement is essentially never inserted in the optimized code. If the detection of the effects of the inserted statement were performed before detecting a created redundant store, we would detect optimizations destroyed by the inserted statement which are actually valid optimizations after the inserted statement is eliminated by redundant store elimination. The process of removing the redundant store would eventually detect that the destroyed optimization is valid and perform the optimization at that time. However, the destruction and then reinstatement of the same optimization can be avoided by careful ordering of detection and update actions.

4.1.2. Optimization dependencies

The design of the incremental optimization algorithms must take into account the dependencies among the optimizations. Compilers usually make multiple passes through the optimization phase to obtain a higher degree of optimization. As statements are eliminated and reordered, conditions for other optimizations become satisfied and are incorporated to produce a more optimized program. Thus, in addition to detection and update of optimizations directly affected by a program change, the algorithms must consider other optimizations affected by the optimized code changes. The extent of this rippling depends on the feasibility of chains of dependent optimizations existing in the optimized code, given the conditions for each optimization.

Creating a redundant store elimination in the optimized code is represented by "covering" the effects of the eliminated statement in MFAD. The covering deletes the statement from the optimized code, but the unoptimized statement remains represented in MFAD. Similarly, in creating a common subexpression the later occurrence of the common subexpression is covered. Destruction of a redundant store elimination must undo the effects of the optimization. This is done by "uncovering" the original unoptimized statement using the model. Uncovering essentially inserts the unoptimized statement into the optimized code at the correct location. However, the statement representation already exists in the model and merely has to be updated for the statement to become part of the optimized code. Uncovering is also involved in the destruction of a common subexpression elimination. Analysis of the covering and uncovering of these optimizations shows the rippling effect of modifying optimizations.

4.2. Incremental Optimization Algorithms

The current algorithms support common subexpression and redundant store elimination and allow program edits which do not affect the flow graph structure. The **insert** and **delete** algorithms are presented in this paper. The **replace** algorithm consists of execution of the **delete** algorithm for the replaced intermediate code statement followed by execution of the **insert** algorithm to incorporate the new intermediate code statement.

The algorithms use the model to detect the effects of program changes disregarding any eliminated redundant store statement (represented by a NOSTORE status) and using the current location of an optimized common subexpression (rather than the original location).

In the discussion of the **insert** and **delete** algorithms in the next two sections, we concentrate on the detection process for affected optimizations. The model and optimized code updates are more fully described for both creation and destruction of each type of optimization in Section 4.2.3. Section 4.2.3 also discusses the rippling effect of changing the code for the affected optimizations. The actual **insert** and **delete** algorithms, given in Figures 3 and 4, include both detection and updates. In the algorithms, **destroycse** and **createcse** are procedures which check for conditions that either destroy or create a common subexpression elimination and then take the appropriate update actions, including recursive calls to the cover and uncover routines. Likewise, **destroyrse** and **createrse** perform similar checks and updates for redundant store elimination. **Covercse** removes the effects of a created common subexpression from MFAD. And finally, **uncovercse** restores the later common subexpression to its original location in response to a destroyed optimization. Calls to **coverrse** and **uncoverrse** are embedded in the create routines for redundant store.

4.2.1. Detection in the insert algorithm

According to the analysis of Section 2, when an intermediate code statement is inserted, a newly validated optimization can be directly created if the inserted statement is (1) a first or later occurrence of a common subexpression or (2) the redundant store or the store causing the redundancy. Similarly, the inserted statement can directly cause a destruction of a redundant store elimination if the statement contains a use of the variable defined in the redundant store and is inserted between the two statements involved in the redundant store. A common subexpression elimination can be destroyed if the statement is inserted between the original locations of the common subexpressions and either (1) a use or definition of the variable defined in the later occurrence of the common subexpression is present or (2) a redefinition of an operand used in the common subexpression occurs.

Therefore, the algorithm for insertion includes detection and appropriate update for each of these situations in addition to proper insertion of the new statement into the model and optimized code. In order to avoid destroying and reinstating valid optimizations during an insertion, we first check if the new statement is a redundant store or the later occurrence of a common subexpression.

If it is a later occurrence of a common subexpression, then the statement is effectively inserted as a simple store after the first common subexpression evaluation. The effects of the operands used in the inserted statement need not be determined, but the effects of the inserted store are checked. If the inserted statement is a redundant store, it is never inserted into the optimized code and its effects need not be checked.

Otherwise the effects of inserting the statement are based on the actual insertion of this statement. Due to the action of the cover and uncover algorithms which handle the rippling of optimizations, these effects can be detected in any order. The **insert** algorithm only considers the inserted statement and its direct affects.

In discussing the actions of the **insert** algorithm, we assume that statement A:=B op C is inserted after statement s-1 in the unoptimized code and thus becomes statement **s**. The algorithm essentially considers all the possible effects of inserting a noncontrol flow statement. In order to determine the correct position of the inserted statement in the optimized code, its child nodes are determined and their parents are examined to determine whether the inserted statement is a later occurrence of a common subexpression. If the children share the same parent node **p** which occurs before the

```
ALGORITHM: INSERT. Insert noncontrol statement.

INPUT:   Statement s before insert-dag node d.
         Statement s' to be inserted.
OUTPUT:  Updated MFAD and optimized code.

NOTE:    STORE status is represented by SS and
         NOSTORE status by NS.

BEGIN {insert}

{determine nodes for uses in s'}
FOR EACH operand i used in s' DO BEGIN {child}
 Let mr=most recent and original def of i<=d with SS;
 {no prior def of i within this basic block}
 IF mr = nil THEN BEGIN {no prior}
  Create leaf labeled i;
  LET mr = leaf created with label i;
  Remove any leaf nodes labeled i after mr;
  END {no prior}
DOEND {child}

{s' contains a variable definition}
IF s' has a def i THEN BEGIN {process def}
 LET mr=most recent def of i <= d with SS;

 {is s' is later occurrence of cse?}
 IF (children share parent p <= d and p has SS) and
  ((mr = nil) or (mr < p and mr has no parent with SS
    between p and d+1)) THEN
  BEGIN {2nd cse}
  Add variable label for i to p; {cse created}
  LET s' map to node p;
  Update node pointer of i at p to record optimization;
  Insert store to i at p;
  END {2nd cse}

 {s' is not later occurrence of cse}
 ELSE BEGIN {create}
  Insert root node n, labels and child pointers;
  Insert expression and store to i at n;
  LET s' map to n;
  END {create}

 LET n = p or n from above; {operator node for s'}

 {check effects of inserting def i at s'}
 {if def of i prior to n, check for created rse}

 LET ns = next def of i after n;
 IF mr <> nil THEN BEGIN {rse check}
  IF original def of i at mr > n THEN
    createrse(n); {rse between cse}
   ELSE BEGIN {not in cse}
    IF mr has no parent with SS < n THEN createrse(mr);
     ELSE IF ns<>nil and mr has no parent with SS
       between n and ns THEN createrse(n);
    END {not in cse}
  END {rse check}
 ELSE BEGIN {no prior}
  IF ns <> nil THEN createrse(n);
  Remove any leaf nodes labeled i after n;
  END {no prior}

 IF def i at s' has SS THEN BEGIN {defeffects}

  {is s' first occurrence of cse?}
  IF there is node f > n with SS and f has
     same uses as s' THEN BEGIN {csecheck}
   LET ms = most recent def of x defined at f < f with SS;
   IF (ms=nil) or (ms < n and ms has no parents with SS
     between n and f) THEN BEGIN {newcse}
    Move variable labels of f to n; {cse created}
    Update child pointers of parents of f to n;
    Update node pointers at n to record optimization;
    Replace expression at f by store x at n;
    covercse(f); {cover later cse}
    Delete node f;
    END {newcse}
   END {csecheck}

  IF mr <> nil THEN updateparents(mr,i,n);
     {cse destroyed if new def between cse}
  END {defeffects}
END {processdef}

ELSE BEGIN {no def i in s'}
 Insert root node n with child pointers from above;
 Insert corresponding optimized code for n;
 LET s' map to n;
 END {no def i in s'}

{check effects of inserting the uses at s'}
IF s' is NOT NEW CSE and NOT RSE THEN
 FOR EACH i used in s' DO BEGIN {process uses}
  LET mr = most recent def of i < n;
  destroycse(mr,i,n,flag); {destroy cse?}
  IF flag THEN BEGIN {cse destroyed}
   LET mr=most recent def of i < mr;
   IF mr <> nil THEN destroyrse(mr,i); {destroy rse?}
   END {cse destroyed}
  ELSE destroyrse(mr,i); {destroy rse?}
 DOEND {process uses}
Update node pointers that reference d;
END. {insert}
```

Figure 3: Insert Algorithm.

---

inserted statement and the two conditions for common subexpression elimination are satisfied, then the optimization is created. A variable label for A is added to **p** and the statement is inserted into the optimized code as a simple store at the first occurrence of the expression. Otherwise, a new operator node **n** is created for A and linked to its previously found child nodes. The statement is inserted into the optimized code at the position indicated by node **n**.

The effects of the inserted statement are then checked according to its location in the optimized code. The algorithm examines the most recent definition of A and next definition of A after s and checks for intervening uses of A to determine whether a redundant store is created. If the statement is inserted between two common subexpressions and A is the variable defined in the later occurrence of the common subexpression, a redundant store is created at s, and thus the common subexpression is not destroyed. However, if redundant store elimination were not included as an optimization, the common subexpression would be

destroyed. If the definition of A in the inserted statement causes a redundant store, the earlier store is covered. If the inserted definition of A is a redundant store, then the remainder of the **insert** algorithm is ignored since the statement is essentially not inserted.

We then check whether B op C is common to a later subexpression by checking the parents of B and C which occur after **n**. If all conditions for optimization are satisfied, the later occurrence of the common subexpression is covered.

In order to completely integrate the inserted node **n** into MFAD, the parents of **n** are determined by examining the parents of the most recent definition of A prior to **n**. This is performed in the procedure **updateparents**. A common subexpression is destroyed if the definition of A at **s** is a redefinition of an operand used in a common subexpression between occurrences of the common expression. This is detected during the update of the parents of **n**. If a parent node, **p**, contains some variable labels with *node pointers* before **n** and some after **n**, then the common subexpression represented at **p** is destroyed.

If the inserted statement is not a common subexpression, the algorithm examines the effects of inserting uses of B and C at statement **s**. A destroyed common subexpression is detected by examining the most recent definition, **mr**, of the operand, say B, prior to **n**. If the original location of **mr** occurs after **n**, then a common subexpression is destroyed by inserting a use of the variable defined in the later occurrence of a common subexpression between the original expression evaluations.

Finally, the status of the most recent definition of B prior to **n** must be examined to detect whether the use of B falls between two stores involved in a redundant store. At this point, the model, mappings, and optimized code are all correctly updated to reflect the inserted statement.

4.2.2. Detection in the delete algorithm

Using the analysis of Section 2, when an intermediate code statement is deleted, a newly validated optimization can be directly created when the deleted statement contains (1) the only use of the variable defined in a redundant store between the two stores involved in the redundant store, (2) a use or definition of a variable defined in the later occurrence of a common subexpression between the expression evaluations, or (3) a redefinition of an operand used in a common subexpression between subexpression evaluations. The deleted statement can directly cause a destroyed optimization when the deleted statement is (1) the one causing the redundant store or (2) the first or later occurrence of a common subexpression. The **delete** algorithm must check for these conditions and subsequently perform the necessary updates.

Similar to the **insert** algorithm, the **delete** algorithm can be improved by avoiding unnecessary checks when the deleted statement is an eliminated redundant store or a later occurrence of a common subexpression. On a redundant store, the model is correctly updated to delete the statement from the optimization history and the remainder of the detection performed in the **delete** algorithm is ignored. When the deleted statement is a later occurrence of a common subexpression, only the effects of deleting its definition are determined. The detection must be done before the statement is actually deleted.

Throughout the following discussion, we assume that the statement being deleted is statement **s** in the unoptimized code which appears as A:=B op C at node **n**. The algorithm checks if a redundant store is destroyed by examining the status of the most recent definition (**mr**) of A prior to **n**. If **mr** has NOSTORE status, then a redundant store to A is destroyed by deleting the store at **s**.

In preparation for the deletion of **s** from the model, the child pointers of each parent of **n** are updated to point to **mr**. Each parent is examined to determine whether the definition of A at **s** is used between two common subexpressions. If two nodes now share the same updated child nodes and all other conditions for optimization are satisfied, a common subexpression is created by covering the later occurrence of the common subexpression.

A created common subexpression is also detected when the most recent definition of an operand (B or C) used in the deleted statement is originally defined after **n**. This indicates that statement **s** uses the variable defined in the later occurrence of a common subexpression between the expression evaluations.

If statement **s** is part of a common subexpression optimization then we update the model and optimized code to correctly undo the optimization. If **s** is the later occurrence of the common subexpression, its variable label is deleted, and if **s** is not an eliminated redundant store, its store is deleted. Otherwise, **s** is the first occurrence of the common subexpression and the common expression must be uncovered at its next occurrence which is not a redundant store. If **s** is not part of a common subexpression, MFAD is updated to delete the subtree representing **s** and the statement is deleted from the optimized code. In this situation, a created redundant store is detected when a child node of **n** is an interior node with no other parent, and there exists a later definition of the variable defined at the child node.

The detection analysis is not performed if the deleted statement is a redundant store. However, updating MFAD to delete the redundant store requires that we distinguish if **s** is a common subexpression. The statement **s** is finally deleted.

4.2.3. Updates for program changes

As program changes are made and optimizations are affected, MFAD must be updated to maintain consistency between the unoptimized code, model, and optimized code. This ensures an accurate optimization history and enables the detection of future affected optimizations in response to a program change given in terms of the unoptimized code. Unlike the detection process, the updating of the representation of redundant stores cannot be ignored. In particular, the child nodes for a redundant store must be kept current, for uncovering the redundant store must find its model subtree representation up to date. The subtree for a covered common subexpression is always current since its subtree represents more than one statement.

The effects of reversing invalidated optimizations and creating new optimizations are handled by a family of cover and uncover algorithms. Any additional affected optimizations must be detected and incorporated into the model. For example, when a redundant store is created, the statement is eliminated in the optimized code and has to be covered in the model. To do this, **coverrse** is called which checks for any created or destroyed optimizations which could occur. By analysis, it can be determined that **coverrse** must check

```
ALGORITHM: DELETE. Delete noncontrol statement.

INPUT:   Statement s to be deleted-dag node n.
OUTPUT:  Updated MFAD and optimized code.

NOTE:    STORE status is represented by SS and
         NOSTORE status by NS.


BEGIN {delete}
Update node pointers that reference n;

{s contains a variable definition--update parents of n}
IF s has def i with SS THEN
 BEGIN {process def}
  LET mr = most recent def of i prior to s;
  IF mr <> nil THEN destroyrse(mr,i); {destroy rse?}

  {i is used in later statements}
  IF n has parents using i THEN BEGIN {parents}
   IF mr = nil THEN BEGIN {no prior def i}
    LET min = lowest parent of n; {1st use of i > s}
    Create leaf labeled i positioned as child to min;
    LET mr = created leaf node;
    END {no prior def i}

  {does i redefine a use in cse between cse?}
  FOR EACH parent p of n using i DO BEGIN
   Update child pointer of p to mr;
   IF mr has a parent q < p with SS
      and same uses as p THEN BEGIN {create cse?}
    LET mp = most recent def of x defined at p < p with SS;
    IF (mp = nil) or (mp < q and mp has no parents<>n
       with SS between q and p) THEN
     BEGIN {newcse}
     Move variable labels of p to q; {cse created}
     Update child pointers of parents of p to q;
     Update node pointers at q to record optimization;
     Replace expression at p by store to x at q;
     coverese(p); {cover later cse}
     Delete node p;
     END {newcse}
    END {check cse}
   DOEND
  END {parents}
END {process def}

{create cse by deleting use of later def of cse?}
IF s is not rse and not later of cse THEN
 FOR EACH operand i used in s DO {for each use}
  createcse(i,s); {create cse?}

{n is a cse node--cse destroyed}
IF n has more than 1 variable label THEN BEGIN {cse node}

 {s is 1st of cse-->i must have SS}
 IF original def of i is n THEN BEGIN {1st store}
  LET ns = next matching cse with SS;
  Update MFAD for any occurrence
   of this cse between n and ns with NS;
  Delete variable label i from n;
  Renumber subtree n to be positioned at ns;
  Move all other variable labels to ns;
  Update child pointers of parents to ns;
  Insert expression and stores at ns;
  uncovercse(statement at n,n,i); {uncover later cse}
  Delete stores at n from optimized code;
  END {1st store}

 {s is later occurrence of cse}
 ELSE BEGIN {later}
  Delete variable label i from n;
  If not rse, delete store to i in optimized code;
  END {later}
END  {cse node}

ELSE BEGIN {n is not cse node}
 FOR EACH child c of n DO BEGIN {child}
  IF c is leaf THEN
   IF c has no other parents THEN Delete c;
   ELSE BEGIN {other parents}
    LET min = lowest parent of c after n;
    Renumber c to become leaf of min;
    END {other parents}
   {c is not leaf}
   ELSE IF c has def j with no other parents
    and there is a later def of j
     THEN createrse(c,j);
   Delete child pointer of n to c;
   DOEND {child}
 Delete node n and its optimized code;
 END {n is not cse node}
END. {delete}
```

Figure 4: Delete Algorithm.

ACTION	EFFECTS			
	Createrse	Destroyrse	Createcse	Destroycse
Uncoverrse	no	yes	yes	yes
Coverrse	yes	no	yes	yes
Uncovercse	no	no	yes	yes
Covercse	no	no	yes	no

Table 1: Relationships of the optimizations.

for created redundant store and common subexpression elimination and destroyed common subexpression elimination. The actions of the cover and uncover algorithms are summarized in Table 1. For example, in covering a statement due to a created common subexpression we need only check for another created common subexpression elimination. The reader can verify these interrelationships by examples or proofs, using the conditions necessary for the existence of each optimization.

Creation of redundant store

When a redundant store is created at statement s, the model is updated to record the new optimization by setting the status of the variable defined at s to NOSTORE and **coverrse** is called for s. In **coverrse**, if s is also the first of a common subexpression, then the model is updated to indicate a destruction of a common subexpression. An algorithm, **uncovercse**, to handle the uncovering of a statement restored by a destroyed common subexpression elimination is called for the second occurrence of the common subexpression. Since the redundant store effectively deletes a statement from the optimized code, each child node is examined. If there are no other parents for that child node, and

the variable defined at the child node is redefined later in this basic block, another redundant store has been created. If so, a new child node is determined for statement s. Covering the operands used in s can also create a common subexpression. Since s is a redundant store, covering the definition at s has no effects.

Creation of common subexpression elimination

Creation of a common subexpression is accomplished by moving the appropriate variable labels and parents' child pointers from the node r representing the later occurrence to node n representing the first occurrence. The *node pointers* of those moved variables are updated to record the original location of the later common subexpression, and the expression evaluation and stores at r are replaced by simple stores at n in the optimized code. The effects of covering the statement at r are examined by the **covercse** algorithm, and the node r is then deleted. Covering the statement can only create additional common subexpressions by deleting the use or definition of the variable defined in a later occurrence of a common subexpression between the common subexpressions.

Destruction of redundant store

When a redundant store to A is destroyed at statement s, the model is updated by setting A's status to STORE and inserting the statement back into the optimized code. The effects of uncovering a redundant store statement are then examined in **uncoverrse**. The newly uncovered statement can create a common subexpression by being the first or last occurrence of a common subexpression which satisfies all conditions for optimization. If so, the model is updated and covering is performed by calling **createcse**. A common subexpression can be destroyed by uncovering the operands used in the redundant store. If so, the procedure **destroycse** is called. Uncovering the operands used in s can also destroy a redundant store elimination by s being located between the two stores involved in a redundant store. **Destroyrse** is then executed again.

Destruction of common subexpression elimination

Destruction of a common subexpression causes the creation of a new root node r at the second occurrence with the same child pointers as the first occurrence at node n. The appropriate variable labels and parents' child pointers are moved from n to r, the simple stores at n are replaced by an expression evaluation and simple stores at r, and the effects of uncovering the later occurrence are examined through the **uncovercse** algorithm. Although the uncovered statement results from a destroyed common subexpression elimination, a new common subexpression can be created if this is the first or later occurrence satisfying all conditions for optimization. The procedure **createcse** is called to update the model and cover the later occurrence. Uncovering the uses in the common subexpression can destroy another common subexpression. The procedure **destroycse** is then executed.

4.3. A Complete Example

As an example of the algorithms, we consider the effects of deleting statement 3 (ie. A=B+C) from Figure 5. Following the **delete** algorithm, we find that the most recent definition of A prior to node 9 occurs at node 3. The NOSTORE status of A at node 3 indicates that a redundant store elimination is destroyed by deleting the definition of A at node 9. At node 3, the NOSTORE status is replaced by a

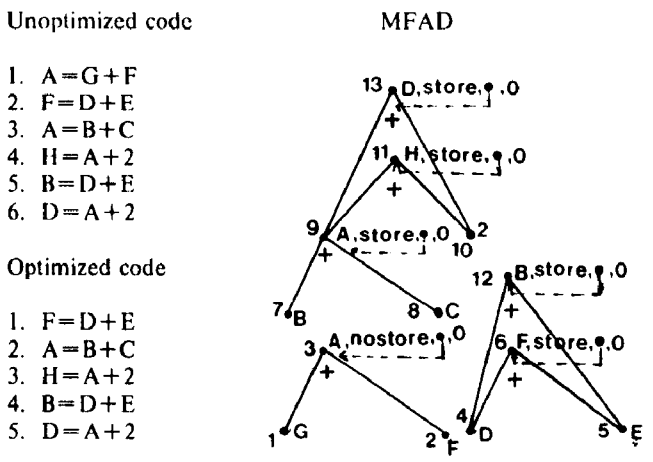

Figure 5: Example of incremental change.

STORE status. A is inserted into the optimized code at node 3, and the uncover algorithm finds that the newly uncovered statement does not affect any other optimizations. The child pointers of the parents of node n, namely nodes 11 and 13, are updated to point to node 3, where the most recent definition of A prior to node 9 occurs. Examining the parents of node 3 reveals that there is no new common subexpression optimization created by the deletion of the definition of A at node 3.

However, when the operands of the deleted statement A=B+C are examined for possible creation of a common subexpression elimination, we see that the deletion of the use of B at statement 3 between the common subexpressions at statements 2 (F=D+E) and 4 (B=D+E) creates potential for a new common subexpression optimization. Since there is no prior definition of B in this basic block, all conditions for optimization are satisfied, and the model is updated by adding a variable label and store for B at node 6 with its *node pointer* pointing to node 11. Node 12 is then deleted with its associated optimized code. The cover algorithm finds that the covering of statement B=D+E creates another common subexpression by covering the use of the variable defined in the later occurrence of the common expression (A+2) between statements 4 (H=A+2) and 6 (D=A+2). The model is updated by adding the variable label and store to D at node 11, deleting node 13, and observing that covering D=A+2 has no other effects.

Since node 9 has only one variable label, it does not represent a deleted common subexpression evaluation. Therefore, each child of node 9 is examined. Since both are leaves with no other parents, they are both deleted from the model. The resulting updated code and model appear in Figure 6.

5. Flow Graph Changes to Locally Optimized Code

The incremental optimization algorithms are currently being generalized to support program edits which affect the flow graph structure. In order to detect optimizations affected by control flow changes, the changes are identified as a sequence of insertions, deletions, or changes to flow graph edges and separations, creations, deletions, or merges of basic blocks. The intermediate code changes which can cause these control flow changes are: (1) insertion of a label on an

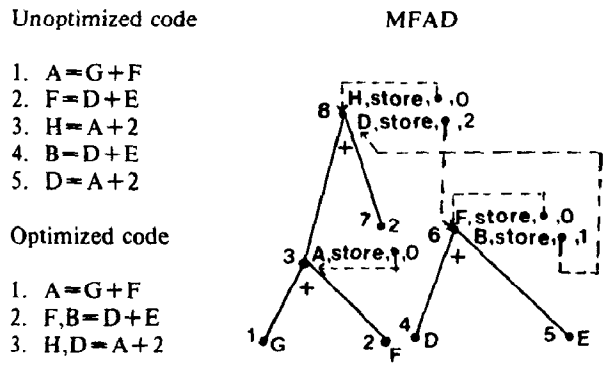

Figure 6: Effects of change on MFAD and code.

existing nonlabeled statement, (2) change of a label on an existing labeled statement, (3) insertion of a labeled statement, (4) deletion of a label on an existing labeled statement, (5) deletion of a labeled statement, (6) change of a target label on an existing control flow statement, (7) insertion of a control flow statement, and (8) deletion of a control flow statement.

These changes can modify the edges of the flow graph, leaving the basic block structure unperturbed, or can actually change the structure of the basic blocks (e.g., merge or split basic blocks). Only those program changes that modify the structure of the basic blocks can potentially invalidate or create newly validated local optimizations since the conditions for performing them are isolated within the given basic block.

Incremental local optimization resulting from the creation of a basic block can be handled as in the initial construction of MFAD. Deletion of a basic block automatically deletes the local optimizations within the basic block without affecting other local optimizations. However, incremental local optimization when merging and separating basic blocks is more complex.

5.1. Merging of two basic blocks

The merging of two consecutive basic blocks might be necessary when a label is deleted from an existing labeled noncontrol flow statement, a labeled noncontrol flow statement is deleted, or a control flow statement is deleted. The merge of two consecutive basic blocks, B and B', into a single basic block is accomplished by inserting each statement subtree of B' successively to the end of B following a postorder traversal of the augmented dag representation of B. The following method for merging basic blocks allows the optimizations currently existing in B' to remain throughout the merge. Each individual merge can be viewed as a special case of the insertion of an unoptimized statement after the last statement of basic block B. Since the statement is currently represented by a subtree in B' and appropriately included in the code, full insertion is not being performed during a merge. For example, realizing that a leaf node in B' represents a value obtained from outside B', only leaf nodes in B' will be replaced by nodes currently in B when there exists a definition of that variable in B.

Created local optimizations are detected and performed as the root of each subtree is merged. If there exists an earlier expression statement in B which matches the statement represented by the subtree currently being merged (i.e., they share the same child nodes) and the conditions for optimization are satisfied, then a common subexpression elimination is performed on the optimized code and the model is appropriately updated. Similarly, a new redundant store is detected by examining whether there exists a use of the newly merged definition after its last definition in B before the merge of this subtree.

As before, covering the statements when creating optimizations by merging can cause further optimizations. However, merging two basic blocks cannot destroy optimizations since optimizations existing prior to the merge were performed locally within each basic block.

5.2. Separation of a basic block

A single basic block might need to be divided into two consecutive basic blocks (B and B') when a label is inserted on an existing nonlabeled statement, a labeled statement is inserted, or a control flow statement is inserted. Given that the program change requires that the basic block B be split after statement s, the dag subtrees representing each statement following s in B are successively updated to be included in the new basic block B' starting with the last statement subtree currently in B and following backwards through the postorder node ordering. Existing local optimizations which are not invalidated by the basic block separation remain in effect in the respective basic block after the separation.

The separation of a given subtree from B is effectively a deletion from B and insertion into B' of the subtree and its associated statements. As each statement is put into B', the effects of deleting its uses from B must be examined. Also, any uses of operands defined prior to **s+1** must be represented in B' by leaf nodes since they now obtain their values from outside B'.

A redundant store optimization is destroyed when the definition of a variable which caused a redundant store resides in B' after separation is complete. A local common subexpression elimination is invalidated when the original location of the later occurrence of a common subexpression resides in B' and the first occurrence resides in B. Changing code to uncover these destroyed optimizations can cause subsequent invalidation of other optimizations. However, local optimizations cannot be created by separating a basic block since no additional sequential code is being constructed which could create potential for further local optimizations.

The procedures for combining and separating basic blocks within MFAD detect newly validated and invalidated optimizations and update the model and optimized code accordingly to represent the new basic block structure, new optimizations, and reversal of destroyed optimizations. The algorithms are special cases of the **insert** and **delete** algorithms, use the optimization history as in the other algorithms, and require no additional information to be maintained. Therefore, supporting program edits which affect the flow graph structure of locally optimized code requires that the current detection/update algorithms be generalized to allow control flow changes, detect the need to merge or split basic blocks, and then execute the procedures outlined above.

6. Extensions to Peephole, Global, and Loop

Detection and update of affected peephole, global, and loop optimizations requires that additional information be integrated into MFAD. Extension of the model to effectively

represent this information and support detection and update of these optimizations is currently being studied. It appears that little additional information is required to maintain an optimization history and enable detection and update of selected peephole optimizations.

Incremental update of unreachable code optimization can be done by observing the number of flow graph edges leading into each basic block. A basic block with no incoming flow graph edges is unreachable and the corresponding segment of code can be deleted from the optimized code. Therefore, when a control flow change affects an existing flow graph edge or inserts a new flow graph edge, the algorithms should check the number of incoming flow graph edges of the involved basic blocks. A previously unreachable basic block becomes reachable when an incoming flow graph edge is added to that basic block, and a reachable basic block becomes unreachable when all incoming flow graph edges of the basic block are removed.

Since an instance of an algebraic simplification optimization is isolated in a single intermediate level expression statement, optimization is automatically destroyed and updated by deletion or replacement of an intermediate code expression evaluation. However, algebraic simplification can occur when an expression statement is inserted. The **insert** algorithm need only be extended to determine whether the inserted expression statement follows one of the supported algebraic simplification laws and insert the corresponding simplified statement in place of the original expression statement. This can be done by keeping tags on algebraically simplified expressions.

A history of a multiple jump optimization can be represented in MFAD by maintaining a list at the node representing an intermediary jump statement **s** which contains pointers to all control flow statements that originally jumped to **s** but have been optimized to skip **s** and go directly to the target label of **s**. The original target label of a control flow statement is kept as a label on the statement's corresponding operator node. Therefore, when a control flow statement is changed, we can identify whether it is an intermediary jump in a multiple jump optimization by the existence of a jump list. We can also determine whether it is the initial control flow statement optimized to jump directly to its final destination by comparing its target label in the optimized code with the target label labeling its operator node. Update consists of updating jump lists and target labels of the appropriate control flow statements as indicated from the detection phase.

Model representations and incremental detection/update algorithms for global and loop optimizations are not yet specified. However, the information that must be saved to enable correct detection and update for these optimizations is outlined here. Constant folding requires the original expression and location of the constant folded expression and current use-definition (ud) and definition-use (du) chain information. The original location and expression of global common subexpressions, a link between the earlier and later occurrence of the common subexpression, and current ud and du chain information are needed to incrementally detect and update global common subexpression eliminations.

Code motion of loop invariants can be represented by a special preheader node which precedes the loop and contains the moved invariant computation with a pointer to its original location within the loop. Current ud and du chain information and detection of loops is also required. Lastly, induction variable elimination requires the original location of the basic and induction variable statements, the values of the basic and induction variables expressed in terms of the temporary variable, current ud and du chain information, and detection of loops.

7. Summary

The ideal incremental programming environment would closely integrate the incremental compiler, incremental optimizer and a symbolic debugger. The similarity of information needed for incremental compilation and the symbolic debugging of optimized code suggests that variations of the same basic internal representation can be used for both purposes.

In summary, we present the first step in a project to develop an incremental optimizing compiler. The model and algorithms presented in this paper can be used to incrementally update locally optimized code. The algorithms modify the optimized code to reflect any destroyed or created optimizations due to the source code changes. They are currently being implemented in C[6] running under UNIX† on a Vax 11/78, and form the basis of a prototype incremental optimization system that we are building.

Besides implementation, future work includes the expansion of the system to include global, loop and peephole optimization. We also intend to examine the construction of an integrated incremental optimizing compiler. Perhaps a concurrent algorithm, similar to that developed by Sawamiphakdi and Ford,[12] can be constructed to speed up the incremental optimization process. Lastly, we intend to evaluate the overhead of incremental optimizing compilers.

A major objection to incrementally compiling optimized code is the additional storage needed during the compilation phase. However, in addition to reduced compilation time due to incremental compilation, execution time and space are decreased by optimizing the code. Greater run-time storage requirements are necessary only when an incremental, optimizing compiler is used in conjunction with a symbolic debugger. In this situation, sufficient information must be present at run-time to allow execution to be stopped, debugging actions to be correctly executed, and desired program changes to be incrementally compiled before execution is resumed. An incremental, optimizing compiler is particularly useful for embedded systems where limited storage may require that code be optimized in order to execute successfully.

References

1. A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Menlo Park, CA (1977).
2. L.V. Atkinson, J.J. McGregor, and S.D. North, "Context-sensitive editing as an approach to incremental compilation," *The Computer Journal* **24**(3), pp.222-229 (1981).
3. Jay Earley and Paul Caizergues, "A method for incrementally compiling languages with nested statement structure," *Communications of the ACM* **15**(12), pp.1040-1044 (Dec 1972).
4. Jeanne Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," RC 10208 (#44947) (August 1983).

†UNIX is a Trademark of Bell Laboratories.

5. Carlo Ghezzi and Dino Mandrioli, "Incremental parsing," *ACM Transactions on Programming Languages and Systems* **1**(1), pp.58-70 (July 1979).

6. S. Graham and M. Wegman, Brian W. Kernighan, and Dennis M. Ritchie, "The C Programming Language," *Journal of the ACM* **23**(1), pp.172-202, Prentice-Hall (1978).

7. John Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems* **4**(3), pp.323-344 (July 1982).

8. Harry Katzan, Jr., "Batch, conversational, and incremental compilers," *Proceedings AFIPS Spring Joint Computer Conference* **34**, pp.47-56 (1969).

9. Raul Medina-Mora and Peter H. Feiler, "An incremental programming environment," *IEEE Transactions on Software Engineering* **SE-7**(5), pp.472-481 (Sept. 1981).

10. Karl J. Ottenstein and Linda M. Ottenstein, "Highlevel debugging assistance via optimizing compiler technology(extended abstract)," *Proc. of ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Highlevel Debugging* (August 1983).

11. Barbara Ryder, "Incremental data flow analysis," *Tenth Annual ACM POPL Conference*, pp.167-176 (January 1982).

12. D. Sawamiphakdi and R. Ford, "A greedy concurrent approach to incremental code generation," *Conference Record of the 12th ACM POPL Conference* (January 1984).

13. Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani, "Incremental compilation in Magpie," *Proceedings of the ACM Compiler Construction Conference* (June 1984).

14. Ron Tischler, Robin Schaufler, and Charlotte Payne, "Static analysis of programs as an aid to debugging," *ACM SIGSOFT/SIGPLAN Symp. on Highlevel Debugging*, Amdahl Corporation (August 1983).

15. Frank Kenneth Zadeck, "Incremental data flow analysis in a structured program editor," *ACM Compiler Construction Conference*, IBM T. J. Watson Research Center (1984).

16. Polle T. Zellweger, "An interactive high-level debugger for control-flow optimized programs," *Proc. of ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Highlevel Debugging* (August 1983).