Incremental Global Optimization for Faster Recompilations

Lori L. Pollock
Dept. of Computer Science
Rice University
Houston, TX 77251
lori@rice.edu

Mary Lou Soffa
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

Abstract - Although optimizing compilers have been quite successful in producing excellent code, their use is limited, due to the accompanying long compilation times and the lack of good symbolic debuggers for optimized code. One approach to attaining faster recompilations is to reduce the redundant analysis that is performed for optimization in response to edits, and in particular, small maintenance changes, without affecting the quality of the generated code. Although modular programming with separate compilation aids in eliminating unnecessary recompilation and reoptimization, recent studies have discovered that more efficient code can be generated by collapsing a modular program through procedure inlining. To avoid having to reoptimize the resultant large procedures, this paper presents techniques for incrementally incorporating changes into globally optimized code. The algorithm determines which optimizations are no longer safe after a program change, and also discovers which new optimizations can be performed in order to maintain a high level of optimization. An intermediate representation is incrementally updated to reflect the current optimizations in the program. The techniques developed in this paper have also been exploited to improve on the current techniques for symbolic debugging of optimized code.

1. Introduction

As generating efficient target code continues to be a major requirement of production compilers, sophisticated code optimization has become an integral part of many language translators. Unfortunately, the longer compilation times associated with optimization analysis and code transformations, as well as the problems of symbolically debugging optimized code, discourage programmers from using optimizing compilers. Optimization is often sacrificed for fast compilation and good debugging capabilities by turning off the optimization phase during program development and exploiting optimization only during the final compilation. However, additional bugs are inevitable, making it difficult to predict the "final compilation". As a result, the programmer is left waiting for slower, optimizing compilations more frequently than desired. Maintenance changes cause long waiting periods while the optimizing compiler regenerates the production code for the modified source program. Bugs not detected by the nonoptimizing compiler are sometimes caught by the optimizing compiler through its extensive program analysis.

One approach to reducing the time for recompilation and reoptimization after a change is to develop techniques to effectively support separate compilation and eliminate the need to recompile the entire set of modules when a single module is edited, despite modules optimized using interprocedural information [4]. With the thrust toward modular programming, research has focused on interprocedural issues with the module or procedure as the smallest unit of reanalysis. However, recent studies have found that better code can be generated by procedure inlining, especially with a register allocation scheme that works well on large procedures, rather than compiling the separate procedures, despite optimization that uses interprocedural data flow information [5,9,17]. Coinciding with these studies, compilers often replace call sites in the intermediate program representation by the bodies of the called procedures prior to optimization, which is typically performed on this intermediate representation. Loop unrolling, often performed to increase the opportunities for optimization of loops, also lengthens procedure bodies. Thus, the sizes of the procedures handled by the optimizer are much larger than the procedures in the source program. A change in what appears to be a small procedure in the source program could in fact be a small change in a very large pro-

* This work was partially supported by the National Science Foundation under Grant CCR-8618699 to Rice University and CCR-8801104 to the University of Pittsburgh.
procedure from the optimizer's view. Reanalyzing the whole procedure or module for optimization to incorporate a code change could involve a significant amount of redundant analysis.

To make optimizing compilation more palatable for the programmer during program development and maintenance, we advocate a fine grain (i.e., statement level) incremental approach to reoptimization in order to further limit the scope of the optimizer to program changes. The optimizer is incremental in that results from previous analyses and transformation are used in an attempt to perform an amount of work proportional to the effect of each program edit. Faster recompilations are achieved by avoiding complete reanalysis of a module in response to a program change. The incremental approach has appeared quite frequently in the literature relating to other facets of compilation [2,6-8,10,16,19]. One of our earlier papers presents algorithms for incrementally incorporating edits into code that is optimized by strictly local optimizations using no information about the flow of data throughout the program [12].

However, there has been no previous work on the incremental reoptimization of globally optimized procedures. By gathering information about the flow of data throughout the procedure, global optimization yields better code than locally optimized code which is performed on individual sequential code segments. Accordingly, incremental reoptimization analysis is much more complicated because a single transformation can affect statements in remote segments of the program. Also, in order to incrementally update a program representation that includes global optimizations, the system must incrementally update global data flow information. A number of incremental data flow analysis techniques have recently been proposed [11,15,18,20].

In this paper, we present a technique for incrementally incorporating changes into globally optimized code. In response to program edits, changes in the conditions for safe optimizing transformations are detected using an annotated intermediate representation of the program and the incremental changes in data flow information. The intermediate representation is incrementally updated to reflect the new optimized version of the program. Since the incremental optimizer reanalyzes only the optimizations that are affected by a program change, updates for small changes can be incorporated at a fraction of the time needed for batch reoptimization of an entire procedure. The bulk of the analysis is performed in response to changes rather than in preparation for possible changes. Thus, analysis is not wasted if, in fact, an edit has no far-reaching effects.

The paper begins in Section 2 by discussing the problems involved in incrementally updating globally optimized code. An overview of our approach to these problems is presented in Section 3. Section 4 describes the annotated intermediate representation used in the incremental analysis. The algorithms that we have developed to perform the incremental reoptimization is presented in Section 5, followed by an analysis of the algorithm in Section 6. Finally, conclusions are discussed in Section 7.

2. Optimization Surgery

In order to preserve the meaning of a source program, each transformation performed by an optimizing compiler should be "safe" in that it does not change the output produced by a program for a given input or cause errors that were not present in the original version of a program. When a program is modified by edits, the conditions for the safety of an optimizing transformation can be altered such that the optimization can no longer be performed without possibly affecting the program semantics. Likewise, there may be optimizations that are safe in the new version, but were not safe in the previous version. These new optimizations need not be performed to maintain correctness, but they must be performed to maintain a high quality of optimized code in response to a series of program changes.

The full effect of a program change involves optimizations directly affected by the user's change and those indirectly affected through the changes in optimizing transformations. The indirect effects of a program edit are caused by the dependencies that exist among optimizations in that performing and removing optimizations can create conditions for other optimizations to become safe or unsafe. The following dependencies could exist between any two optimizations \( \alpha \) and \( \beta \):

- performing \( \alpha \) establishes the safety of \( \beta \)
- removing \( \alpha \) establishes the safety of \( \beta \)
- removing \( \alpha \) destroys the safety of \( \beta \)

The dependencies with \( \alpha \) and \( \beta \) interchanged could also exist. It should be noted that performing an optimization can never destroy the safety of another optimization because an optimization is never performed on the premise that another optimization will be reversed to make it safe.

Figure 1 presents an example of an optimization dependency graph, which is our graphical representa-
tion of the full effect of a particular program change on optimizations in terms of the dependency relationships that actually lead to each optimization reversal or creation. Each node (except the root that represents the original edit) represents the creation or reversal of an optimization performed sometime during the incremental update. For example, node 2 (destroy cs d+e) in Figure 1 represents the destroyed safety and reversal of the common subexpression optimization of 1: x:=d+e; and 4: y:=d+e; to x:=d+e; y:=x; by insertion of d:=m+b.

A directed edge from node n to node m in the update dependency graph indicates that the transformation represented by node m depends on the prior transformation represented by node n. For example, the edge from node 2 (destroy cs d+e) to node 3 (destroy cp y=x) indicates that the propagation of the definition of y in statement 4 to all of its uses and then the elimination of the copy y:=x depends on the common subexpression optimization of d+e that created the copy. When the common subexpression is removed in response to the edit, the copy propagation is no longer applicable. The paths leading into a node n in the optimization dependency graph depict the dependencies of all prior code changes leading to the optimization change (i.e., creation or reversal) at node n. Only the last change along a path to a node actually triggers the new optimization change. The solid edges in Figure 1 indicate the triggering dependencies.

Assuming that optimization changes are detected sequentially, an ordering (i.e., the order in which these changes are detected) is imposed on the optimization reversals and creations in response to an edit. Each node of the optimization dependency graph is labeled by an integer indicating the sequential ordering of the optimization changes imposed by the incremental optimizer. If this sequential ordering of changes is altered, a different optimization dependency graph will result. Since a change can not be caused by a later change, the optimization dependency graph is acyclic (i.e., a directed acyclic graph, or dag).

Another problem in developing an incremental optimizer concerns the mapping from source to target code. Similar to a symbolic debugger, a fine grain incremental compiler must maintain a mapping from source program constructs to the corresponding generated code segments in order to detect statements that are affected by a program edit and then reanalyze only the affected statements. The main task of the incremental optimizer is to update globally optimized code to reflect the changes in the safety of optimizing transformations in response to program changes. However, its task is complicated by the effect of optimizations on the mapping required by an incremental compiler. Code optimizing transformations complicate this mapping in three ways: (1) Optimizations such as copy propagation and elimination suspend the generation of code for portions of source statements. (2) The generated code may be rearranged by optimizations such as loop invariant code motion where a computation that is invariant to the loop's execution is moved to a location just prior to the loop entry. (3) A statement may be replaced by another statement or sequence of statements. For example, global common subexpression elimination replaces expression evaluations by simple copies. These effects must be considered, and in fact, exploited in the design of an incremental optimizer.
3. An Event Driven Approach

Our approach to incremental reoptimization is to explicitly detect which optimizations become unsafe and which new optimizations can be performed in response to a code change. As batch optimization gathers information about the flow of data at different program points and uses this information to determine when an optimizing transformation can be performed safely, an incremental approach to reoptimization must also utilize global data flow information. Our algorithm for incremental reoptimization is actually driven by the changes in data flow information. In particular, the optimization dependencies implicitly direct the incremental reoptimization through the changes in data flow information. Both program edits and code transformations from affected optimizations trigger incremental data flow analysis. Likewise, data flow changes trigger the detection of changes in conditions for the safety of optimizing transformations and the incremental update of the intermediate representation. The updates include the incorporation of the new optimizations and the exclusion of the old optimizations that have become unsafe. Only blocks where data flow changes have occurred are considered as candidates for possibly affected optimization.

Data flow changes and an updated program representation are sufficient for detecting new safe optimizations. However, in order to use data flow changes to determine which existing optimizations have become unsafe, a record of each existing optimization is embedded into the intermediate program representation. When an optimization is performed, the intermediate representation is annotated to indicate how the transformation has affected the current code. After detecting conditions that invalidate the safety of an optimization, the record of the optimization is removed from the intermediate representation since it no longer has an effect on the optimized code.

4. Annotated Intermediate Representation

Invoked between semantic analysis and final code generation, an optimizer typically performs its analysis and transformations over a control flow graph representation of the source program. The data dependencies among statements in each sequential code segment (i.e., basic block) can be represented as a directed acyclic graph (i.e., dag). With the goal of improving the generated code, an optimizer transforms this intermediate representation without retaining any information about the individual optimizations that were performed. Figure 2 shows both the flow graph representation and the linearized intermediate code for the optimized version of a small source code segment. We conservatively assume that temporaries and program variables may be used later in the program. The expression \( j+10 \) is folded into the constant 15 and moved outside the loop. The common subexpression evaluation \( i-l \) is replaced by the value of \( n \) at labeled statement 12, and the references to variable \( m \) are replaced by the constant 15.

The original expressions and variable references are not retrievable from the optimized intermediate representation. Similarly, the original location of the loop invariant \( m=15 \) cannot be discerned after optimization. Since the incremental reoptimization scheme needs to identify individual optimizations in order to determine whether they remain safe after a program change, we retain the subtrees from the original, unoptimized code and annotate the dag nodes of these subtrees. With these small modifications to the optimizer, our intermediate representation simultaneously represents the unoptimized and optimized program code with an indication of the individual machine independent optimizations responsible for the current form of the optimized code [12]. The optimized code can be generated from this annotated intermediate representation by traversing the control flow graph with a postorder traversal of each dag (depicted in the figures by the integer labels on each dag node), using the annotations to direct the optimization. The unoptimized code can be generated by the same traversal, but using the annotations to regain the effect of the original code.

The annotations to the intermediate representation are dictated by the information required to maintain a complete snapshot of existing optimizations, adequate to determine when optimizations become unsafe due to edits. Thus, the annotations depend on the optimizations being supported by the incremental compiler. However, the information recorded for a transformation can be generalized. Typically, the annotations must represent code that has been eliminated, relocated, or replaced by optimization. Thus, a variable label for a variable \( v \) on a dag node is extended to include (1) a field indicating whether the value represented by the node is actually stored to \( v \) (store) or the store to \( v \) has been eliminated by optimization (nostore), (2) a field consisting of a pointer to a dag node and an index into the list of labels on that dag node indicating the location of the original store to \( v \), and (3) a pointer to references to which the constant value of \( v \) has been propagated. Since the order in which existing optimizations are affected in response to edits is independent of the order that they were ini-
```plaintext
Source | Unoptimised | Optimised
--- | --- | ---
... | ... | ...
j := 5; | j := 5 | j := 5
i := 0; | i := 0 | i := 0
repeat 11: i := i + 1 | m := j + 10 | 11: i := i + 1
i := i + 1; | m := j + 10 | 11: i := i + 1
m := i + 1; | if n > m | if n > 15
if n <= m | goto 12 | goto 12
then print(m) | print(m) | print(15)
else print(i+1) | goto 13 | goto 15
until (i >= 100),
... | ... | ...

Figure 2. Intermediate program representation.
```

In order to detect affected global optimisations using data flow changes, we maintain current data flow information at each basic block as well as the data flow information that was current when the block was last examined for affected transformations. For a specific data flow problem, the updated GEN, KILL, IN, and OUT sets and the IN and OUT sets in effect before the current change are saved at each basic block. For a basic block B, GEN(B) contains information about data generated within B, while KILL(B) contains information about data that is stopped from flowing upon executing B. IN(B) summarizes the flow of data at the start of B, while OUT(B) summarizes the flow of data on exit from B.

In this paper, we refer to reaching definitions data flow information. Reaching definitions at a point...
p in a program are the definitions of variables that reach p with no intervening redefinition. They can be used in determining the safety of constant folding, copy propagation, and loop invariant code motion.

5. The Incremental Reoptimisation Algorithm

Figure 4 presents the algorithm used in incremental reoptimization. Incremental analysis begins by detecting the affected optimizations within the basic block containing the program change. We refer to these optimizations as the locally affected optimizations. The annotated intermediate representation is updated to reflect these transformations. The nonlocalized nature of global optimizations suggests that update for a single global optimization can transform multiple blocks such that intermediate code for several blocks must be updated to completely record the transformation. In order to reduce the frequency of incremental data flow analysis, we detect any additional optimizations within each block transformed by these locally affected optimizations before invoking the incremental data flow analyzer.

The incremental data flow analyzer is then invoked to incrementally update the data flow information throughout the program based on the changes in the GEN and KILL sets at any block transformed by the locally affected optimizations. The incremental reoptimization algorithm is designed to be completely independent of the algorithm used for incremental data flow analysis, so any of the available techniques can be employed [11,15,18,20]. The incremental data flow analyzer returns with a set identifying the blocks where data flow changes have occurred. This set becomes the initial value of the worklist of basic blocks that need to be examined for potentially affected optimizations.

When a block B is removed from the worklist, the affected optimizations at B are determined based on the changes in the data flow sets IN(B) and OUT(B). As optimizations are detected as being affected, the annotated intermediate representation is updated. After detecting all of the optimizations affected within a block B, the data flow sets that record the previous global data flow information at B are updated to the values of the current data flow sets at B. The global effects of the transformations are propagated by again invoking the incremental data flow analyzer to update the data flow sets throughout the program. The set of blocks with data flow changes, returned by the incremental data flow analyzer, is added to the current worklist, and the update process continues. Incremental reoptimization terminates when there are no more blocks with global data flow changes, reflected by an empty worklist.

5.1. Locally Affected Optimizations

Figure 5 demonstrates the analysis for determining the locally affected optimizations in the block containing the program change. We show the analysis for the effects of inserting a variable definition on global common subexpressions and constant folding. The remove and perform algorithms for each optimization update the annotated intermediate representation for the transformation and determine any additional local effects of the transformation. The remove algorithms remove any trace of the optimizations from the annotated intermediate representation, while the perform algorithms record the transformations appropriately. These algorithms also update the corresponding GEN and KILL information and recur to detect and update other locally affected optimizations based on optimization dependencies.

5.2. Remote Affected Optimizations

In Figure 6, we illustrate the detection analysis performed at each basic block where global data flow information has changed. We refer to these blocks as remote blocks. The algorithm is executed for a block B when B is removed from the worklist for detection of affected optimizations. In particular, we present the analysis for constant folding at a remote block. Affected constant fold optimizations are identified by

```plaintext
INPUT: Inserted statement in basic block B.
OUTPUT: Updated annotated intermediate representation.
DECLARE worklist: basic blocks with altered data flow.
BEGIN
    worklist := {};
    REPEAT
        IF (worklist is not empty) THEN BEGIN
            FOR each block B with altered GEN(B) or KILL(B)
                DO BEGIN
                    Remove a basic block B from worklist;
                    update_remote_affected_optimizations(B);
                    old_data_flow_sets(B) := current_data_flow_sets(B);
                    IFEND;
                    FOR each block B with altered GEN(B) or KILL(B)
                        DO BEGIN
                            Update IN(B) and OUT(B);
                            worklist := worklist \ incremental_data_flow(B);
                        DOEND;
            END;
        UNTIL worklist is empty;
    END.

Figure 4. Incremental reoptimization algorithm.
```
INPUT: Inserted definition of variable v at statement s, represented by dag node n, in block B.
OUTPUT: Updated annotations for common subexpressions (ca) and constant folding (cf).

BEGIN (common subexpressions)
IF (there is no prior definition of v in B) THEN
FOR EACH expression e in B represented by a subtree with n as a new child node DO
IF (e is marked later of ca) THEN remove_ca(e);
END.
END

FOR EACH parent p of n DO BEGIN
IF (p is marked cf) THEN remove_cf(p),
IF (new definition is constant or marked cf) AND (all other child nodes of p are constant, single constant definition or marked cf)
THEN perform_cf(p),
END;
END.

Figure 5. Analysis of locally affected optimizations.

5.3. Loops and Loop Optimization

Changes in a program's control flow structure can affect the program's loop structure, indirectly affecting loop optimizations through loop creation, destruction, expansion, and contraction. Each natural loop is a single entry, strongly connected region of the flow graph and has an associated back edge, namely an edge a→b where b dominates a. A node n dominates another node m in the flow graph if every path from the initial node of the flow graph to m passes through n. A loop structure is destroyed when its back edge is destroyed either by deleting the back edge itself or destroying the dominator relation between the head and tail of the back edge. A new loop is created when a back edge is created by either inserting the back edge or creating the necessary dominator relation between the head and tail of an edge.

A loop can be reduced in size without completely destroying the loop structure by deleting an edge with both head and tail inside the loop such that the source of the loop has no more edges leading into the loop and the loop's back edge remains. An existing loop can be expanded when an edge is inserted to create a path of nodes with an ancestor and descendant inside the loop and no edges from outside the loop into any node along the new path. Loop contraction and expansion can also be caused by deleting or inserting a back edge that shares a header with another back edge when the associated sets of loop nodes are not a subset or superset of each other.

The analysis for loop structure changes and affected optimizations is illustrated in Figure 7. By examining the dominator relation between the head and tail of each inserted edge, inserted back edges are easily identified. By keeping a record of all existing back edges, deletion of a back edge is easily detected. Changes in dominators are used to detect other destroyed and created back edges. Dominator information is incrementally updated when an edge is inserted or deleted, and the dominators at nodes with dominator changes are examined [3,13].

When a new back edge is identified, the set of nodes that form the corresponding loop are determined by the node stacking algorithm used in traditional optimizing compilers [1]. The loop's relationship with other loops (i.e., superset, subset, or disjoint) is determined in order to keep a record of the loop nesting.
6. Analysis of Incremental Reoptimization

In this section, we discuss the correctness, effectiveness, and efficiency of incremental reoptimization. In order to show that the incremental reoptimization algorithm is correct, we show that the following two properties hold: (1) The incremental reoptimization algorithm terminates. (2) The functional equivalence between the optimized and unoptimized code is preserved after a program change is incrementally compiled [13].

Since the incremental scheme is guided primarily by program edits and optimization dependencies (indirectly through data flow changes), incremental optimization may not produce the same optimized code as a multiple pass, batch optimizing compiler. However, it can be shown that the incremental optimization analysis is complete in that there are no more optimizations (of the kinds supported by the optimizer) that can be performed on the current code when the algorithm terminates, assuming that there were no more possible optimizations before edit. The proof of this property follows closely to the proof of functional equivalence [13].

Similar to the batch scenario, the order of detection and update of affected optimizations can affect the quality of the optimized code. In general, the order of updates is driven by the worklist. However, when detecting the local effects within a basic block, the order of updates can be based on the relative effectiveness of different optimization orders.

Driven by the worklist of basic blocks with data flow changes, incremental reoptimization analysis is performed only at basic blocks where data flow information has changed and at the basic block containing the program edit. This may include blocks where no optimizations are affected since the optimizations within a block may not be affected by the changes in data flowing into the block, but no updates or incremental data flow analysis will be performed when optimizations in a block are unaffected. For each block B removed from the worklist, the time accounted to processing B is roughly

\[(a + \beta + \gamma) \cdot \text{(number of blocks T)}\]

where

- T = a block transformed directly by B's optimizations.
- \(a = \text{time to detect affected optimizations at T}\)
- \(\beta = \text{time to update T's annotations, GEN(T), KILL(T)}\)
- \(\gamma = \text{time to globally update IN and OUT sets based on changes to GEN(T) and KILL(T)}\)
The value $\alpha$ depends on the data flow changes, the existing optimization of $T$, and the opportunity for optimization of $T$. Since it takes constant time to update the annotated intermediate representation and GEN and KILL to reflect an optimization update, the value $\beta$ is a constant times the number of optimizations affected in $T$. The value of $\gamma$ depends on the algorithm used for incremental data flow analysis. We expect that the number of blocks transformed directly by a block's affected optimizations (i.e., the number of blocks $T$ for a given block $B$) will be small. The number of blocks $B$ removed from the worklist throughout incremental update depends on the size of the edit, program structure, and optimizations in the code. Since worst case scenarios are not of practical interest in an incremental setting, it is difficult to assess the actual speedup of recompilation due to incremental reoptimization without empirical studies, which are currently underway.

Control flow changes are the most expensive edits to handle as they can affect the loop structure of the program as well as the flow of many different variables. The incremental update of data flow is more costly, and the number of affected optimizations is potentially greater due to the larger impact of the change. Changes to expressions and variable definitions or uses are incorporated with less effort since the effect on reaching definitions is limited to a single variable and loop structures are unaffected. With the exception of global common subexpressions, the merge, separation, deletion, or insertion of basic blocks has no effect on global optimizations since the flow of data throughout the flow graph remains unchanged.

In addition to improving the quality of the target code, careful ordering of incremental updates can improve the efficiency of incremental reoptimization by avoiding the destruction and reinstatement (or vice versa) of the same optimization. Thus, our deletion algorithm begins by identifying whether the deleted statement currently exists in the optimized code, and if so, determines its current form and location. If the deleted statement was previously eliminated by optimization, no further detection is required because deleting a statement that does not exist in the optimized code has no effect. Similarly, if optimization had replaced the statement by a simpler statement, then less analysis is needed to detect the effects of deleting the current, less complex statement. If the statement is moved by an optimization, the effects of deletion are determined according to its current location rather than the original one to avoid unnecessary analysis. Similarly, those optimizations that move, replace, or eliminate an inserted statement are detected early in the insertion algorithm.

For incremental reoptimization, the annotated intermediate representation and pertinent data flow information must be available throughout editing. Since this information is not needed during execution unless utilized for symbolic debugging, it need not be readily accessible until the programmer returns to editing and incremental compiling. Since the annotations are designed to record the generalized actions of optimizations, namely elimination, reordering, and replacement, the size of the annotated intermediate representation should not increase in great proportions as new optimizations are supported by the incremental optimizer. Moreover, most of the data flow information and the intermediate representation can be shared by other software tools commonly found in a programming environment.

7. Conclusions

We have presented a technique for reducing the long compilation times typically associated with ambitious optimizing compilers. Based on annotating the intermediate representation typically used during optimization and incrementally processing intermediate code changes, the incremental reoptimization algorithm can be easily incorporated into an incremental compiler.

In a separate paper, we describe how we have exploited the incremental reoptimization analysis, annotated intermediate representation, and data flow information to improve upon the current techniques for symbolic debugging of optimized code [14]. The incremental reoptimization analysis could also be exploited in other contexts. The event driven approach offers an alternative to the traditional multipass optimizer. Additional optimizations could be discovered after an initial optimization pass based on optimization dependencies implicit in the changes in data flow information. Incremental reoptimization analysis could be easily extended to handle optimizations that have been performed exploiting interprocedural data flow information. Edits in one module would result in changes in the interprocedural information at various call sites in other modules. Our incremental reoptimization analysis could interpret these interprocedural changes as local data flow changes at the blocks containing the call sites.

The incremental analysis can be used to provide precise identification of modules that require recompilation in a separate compilation environment which supports interprocedural analysis and optimization.
Current techniques are either approximate or very expensive [4]. These environments also schedule modules for recompilation only when the indicated changes in the interprocedural information cause existing optimization to become unsafe. Using our analysis, conditions permitting additional optimizations can also be detected and thus modules can be earmarked for recompilation in order to maintain highly optimized code. Ideally, the information maintained for incremental optimization analysis could aid in predicting the impact of changes on a module, and this information could be exploited by the recompilation system to make intelligent decisions on whether the module would be more efficiently recompiled at a statement level using incremental reoptimization or completely recompiled.

References


