

# Interprocedural Optimization: Eliminating Unnecessary Recompile

MICHAEL BURKE  
IBM Research  
and  
LINDA TORCZON  
Rice University

---

While efficient new algorithms for interprocedural data-flow analysis have made these techniques practical for use in production compilation systems, a new problem has arisen: collecting and using interprocedural information in a compiler introduces subtle dependences among the procedures of a program. If the compiler depends on interprocedural information to optimize a given module, a subsequent editing change to *another* module in the program may change the interprocedural information and necessitate recompilation. To avoid having to recompile every module in a program in response to a single editing change to one module, we have developed techniques to more precisely determine which compilations have actually been invalidated by a change to the program's source. This paper presents a general *recompilation test* to determine which procedures must be compiled in response to a series of editing changes. Three different implementation strategies, which demonstrate the fundamental tradeoff between the cost of analysis and the precision of the resulting test, are also discussed.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Interprocedural analysis and optimization, data-flow analysis, recompilation analysis

---

## 1. INTRODUCTION

Traditional optimizing compilers have advanced to the point where they do an excellent job of optimizing code within a single procedure or compilation unit. Accordingly, code optimization research has begun to focus on interpro-

---

This research has been supported by the IBM Corporation, by the Advanced Research Project Agency, and by the National Science Foundation through grants MCS 81-21844 and MCS 83-03638.

Authors' addresses: M. Burke, IBM T. J. Watson Research Center, POB 704, Yorktown Heights, NY 10598. email: burkem@watson.ibm.com; L. Torczon, Computer Science Dept., Rice University, Houston, TX 77251-1892. email: torczon@rice.edu

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1993 ACM 0164-0925/93/0700-0367 \$01.50

ACM Transactions on Programming Languages and Systems, Vol 15, No 3, July 1993, pages 367-399.

cedural analysis and optimization. Recent work has included both faster algorithms for some interprocedural data-flow problems and efficacy studies of some interprocedural transformations [7, 8, 9, 13, 15, 16, 24, 25, 33]. These techniques are also being applied in compilers that try to automatically restructure programs to expose parallelism. In each of these areas, the goal is to improve the efficiency of code generated for a whole program by giving the compiler more context over which to optimize.

Unfortunately, interprocedural optimization directly conflicts with one of the most treasured features of traditional ALGOL-like programming languages: separate compilation. Interprocedural data-flow analysis gives the compiler facts about the naming environment in which the code will execute at run-time and about the side effects of procedures that will be called at run-time. Using such information makes the correctness of compile-time decisions for one procedure dependent on the source text for other procedures. Cross-procedural optimizations, like interprocedural register allocation and inline substitution, have a similar effect, although they may rely on information derived even later in the compilation process, like the specific mapping of names to storage locations. As soon as information from other procedures is used to make compile-time decisions, the object code produced by the compiler becomes a function of those other procedures. In such a system, editing changes made to one procedure can invalidate prior compilations of other procedures.

To produce a practical system that performs interprocedural analysis and optimization will require a mechanism for tracking such recompilation dependencies, detecting when they are violated, and automatically recompiling the necessary parts of the program. Of course, the compiler could adopt the naive approach and recompile the entire program after a change to any single procedure—sacrificing any possible benefit of separate compilation. The alternative is to perform a *recompilation analysis* to determine, at compile-time, the set of procedures that may need to be recompiled in response to editing changes to one or more procedures in the program. The power (and success) of such an analysis should be measured by the number of spurious recompilations that it avoids—procedures that would otherwise be recompiled unnecessarily.

This paper examines the recompilation problems introduced by the use of interprocedural dataflow information as a basis for compile-time decisions. It extends the work presented by Cooper et al. [17] and Torczon [32]. We present a general approach to recompilation analysis and three specific techniques for implementing it. The general framework is based on observing the nature of the interprocedural sets themselves and the ways in which an optimizer can use them. The different implementation techniques produce recompilation tests of successively greater precision, with a concomitant increase in the expense of the test. Each of the techniques represents a significant improvement over recompiling the entire program.

This problem has not received much attention in the literature, primarily because few compilers have actually computed and used interprocedural information. For example, the PLI Optimizing Compiler trivializes the prob-

lem by limiting its analysis to a single compilation unit [28]. Other systems, like the ECS project at IBM Research, appear to recompile the entire program for each executable [3]. Some systems ignore the problem completely—for example, the Cray FORTRAN compiler will perform inline substitution, but does not provide the user with support for managing the resulting recompilation problems.

There are systems that are related to this work. Feldman's *make* system is an ancestor of our system. It pioneered the idea of automatic reconstruction based on an analysis of the internal dependences of a system [20]. However, *make* requires that the programmer explicitly specify the compilation dependences, while our method derives them analytically. The system proposed by Tichy [31] and Tichy and Baker [30] analyzes the recompilation dependences introduced through the use of *include files*. For each module that uses a specific include file, it records those definitions that are actually referenced in the module. Using this information, it can determine which modules must be recompiled when an include file is changed. Schwanke and Kaiser suggest that the number of routines selected for recompilation by the Tichy-Baker approach could be further reduced if harmless inconsistencies between separately compiled modules were not considered when determining which routines to recompile [27]. Although these systems are similar in flavor to our approach, they do not determine if changes in the interprocedural data-flow information for a procedure make recompilation of that procedure necessary.

Although the implementations of this work have been in the context of systems that analyze FORTRAN, the techniques are applicable across a wide variety of languages. They work directly with data-flow information produced by a compiler; the complications introduced by specific language features are thus folded into computing the base information on which our methods work.

The remainder of this paper is subdivided into eight sections. Section 2 describes our model of the compilation system. Section 3 introduces the three specific kinds of interprocedural data-flow information addressed by our work. Section 4 describes the general recompilation framework and presents three instantiations of the framework. Section 5 proposes a more elaborate instantiation of the framework that can lead to more precise recompilation tests. Section 6 discusses optimizations that directly use interprocedural facts. Section 7 generalizes the work to deal with multiple procedures in a single compilation unit and to account for the effects of interprocedural optimizations. Section 8 addresses the dual of the problem—predicting when a recompilation might be desirable to improve the results of optimization. Finally, Section 9 contains a summary and some conclusions.

## 2. COMPILATION MODEL

To simplify the remainder of this discussion, we will first present a model of the compilation system. The techniques that we describe in this paper are intended for use in a compiler that attempts both separate compilation and the collection and use of interprocedural data-flow information. Such a compiler must be structured differently than one that supports a traditional

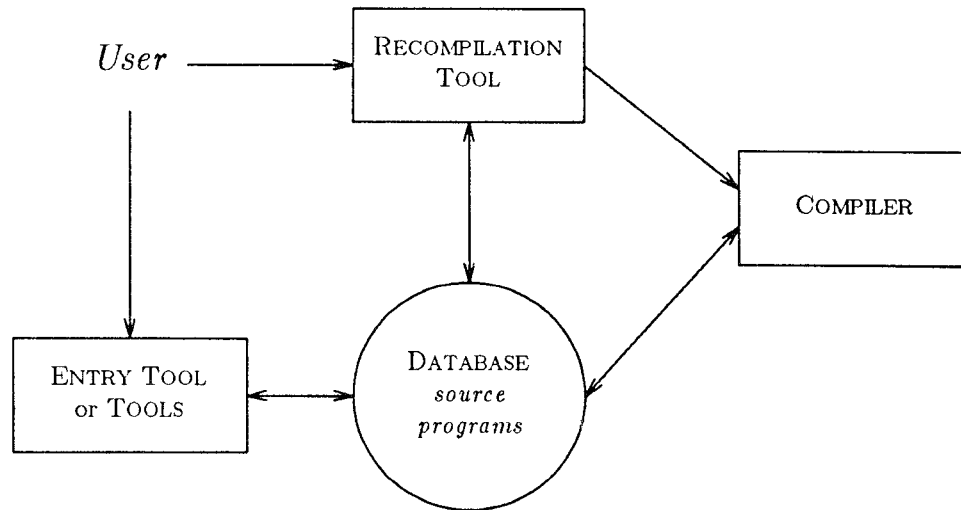


Fig. 1. Our compilation model

separate compilation scheme. The differences arise from two principal requirements:

- (1) The compiler must have access to information about all the procedures in a program as it compiles each of them.
- (2) The compiler must have the ability to “retract” optimizations, after the fact, in response to changes in the interprocedural information that was used to justify them.

Together, these observations suggest a new relationship between compiler, source code, and programmer, depicted in Figure 1.

The entry tool provides the programmer with a means to create and modify source text that resides in the system’s database. The entry tool can be implemented as a language sensitive editor or some combination of editor, version control system, and compiler front-end. A similar path must allow the programmer to construct a representation of the program—a recipe that specifies how to bind together individual source code modules to form a single executable program. The recompilation tool creates an executable image for this program. It uses information from the database to determine what must be compiled and uses the compiler and linker as needed. The compiler has direct access to the database for interprocedural information, as well as the results of previous compilations. We assume that the compiler translates only one procedure at a time; in Section 7, we show how to extend the recompilation tests to larger compilation units.

The techniques described in this paper have been designed to operate in a system structured like our model. The model is not overly restrictive. It can accommodate an implementation in the context of a programming environment—the  $\mathbb{R}^2$  programming environment for FORTRAN is an example [10].

Similarly, an implementation within a more conventionally structured compiler can fit the model—the PTRAN analysis system for FORTRAN is an example [2]. Both of these systems implement recompilation analysis using techniques described in this paper.

### 3. INTERPROCEDURAL INFORMATION

Familiarity with interprocedural data-flow information is a prerequisite to understanding the recompilation tests, so we begin with some background. Interprocedural information provides the compiler with knowledge about the run-time conditions under which a procedure will actually be invoked and about the impact of executing other procedures on the run-time values of variables in the procedure being compiled. We are concerned with three distinct interprocedural phenomena: aliasing, side effects, and constant propagation.

*Aliasing* occurs when two or more names, at some point in a program, refer to the same storage location. Because an assignment actually modifies both the name and all of its aliases, the compiler needs reasonably precise information about aliases.<sup>1</sup> We limit our consideration to the *interprocedural aliases* generated by the call-by-reference parameter mechanism. For example, when a variable  $v$  is passed by reference at a call site to a formal parameter  $x$ ,  $v$  and  $x$  become aliases of each other if  $v$  is also visible inside the called subroutine. Two variables are *potential aliases* in procedure  $p$  if they refer to the same storage location in some execution instances of  $p$  (i.e., the variables are aliased along some, but not necessarily all, execution paths leading to  $p$ ). The compiler can compute, for each procedure  $p$ , a set  $\text{ALIAS}(p)$  containing those pairs of names that are potentially aliased in  $p$  [5A, 14]. In the absence of such information, the compiler must assume that all formal parameters and global variables are potentially aliased. In practice, this eliminates opportunities for optimizations involving those variables.

*Side-effect summary information* describes the effects of executing a procedure call on the values of variables. At a call site, executing the body of the called procedure can both reference and change the values of individual variables. Since the compiler relies on derived knowledge about the values of variables to determine the safety and profitability of optimizations, the impact of a procedure call on the values of variables in the calling procedure must be considered. The compiler uses this information to sharpen its analysis within a single procedure. In the absence of precise information, the compiler must assume that the call both modifies and uses every variable available to it. Using such worst case assumptions decreases the accuracy of the data-flow information computed for the calling procedure, potentially inhibiting optimization within that procedure.

<sup>1</sup>Strictly speaking, the FORTRAN standard permits the compiler to ignore aliasing. The standard contains a restriction that neither of the two aliases may be modified in a standard-conforming program [5]. Nevertheless, many compilers attempt to trace aliases because information about potential aliases can be useful as a diagnostic aid to the programmer and because the resulting systems achieve a higher level of predictability than the standard requires.

In our model system, the compiler will annotate each call site  $e$  in a program with two sets,  $\text{MOD}(e)$  and  $\text{REF}(e)$ .<sup>2</sup> The former contains all variables that might be modified as the result of executing  $e$ , while the latter contains all those variables whose values might be referenced as the result of executing  $e$ . For example, in traditional available expression analysis, a procedure call must be assumed to kill every expression involving either a global variable or an actual parameter. But if the compiler encounters an expression  $v$  that is available immediately before call site  $e$  and it determines that none of the constituent variables of  $v$  are in  $\text{MOD}(e)$ , then it can assume safely that  $v$  is still available after  $e$ .

In large programs, information is often passed between procedures in the form of constant-valued actual parameters or global variables. This is particularly common in numerical programs that incorporate modules from standard libraries such as LINPACK [19], and in programs where the dimensions of major data structures are stored in variables to simplify later modification. *Interprocedural constant propagation* attempts to identify formal parameters and global variables that will have the same known constant value on each invocation of a procedure within a given program. Finding a precise solution to the general constant propagation problem is undecidable [22] and the usual approximate constant propagation problem is intractable in an interprocedural setting [23]. However, a useful subset of the complete and precise set of interprocedural constants can still be profitable for the optimizer. The algorithms for this problem proposed to date compute approximations to the sets of constant-valued parameters and global variables [7, 9, 32, 34]. In our model system, the compiler computes, for each procedure  $p$  in the program, a set  $\text{CONSTANTS}(p)$  of constants known to hold on entry to  $p$ . Elements of  $\text{CONSTANTS}(p)$  are pairs of the form  $(x, v)$ , where  $x$  is the name of a formal parameter or global variable and  $v$  is its known constant value.

As an example, consider the program fragment shown in Figure 2. Assuming that all of the relevant statements are shown, the aliasing and constants sets for its procedures would be:

<i>Procedure</i>	<i>ALIAS</i>	<i>CONSTANTS</i>
a	$\emptyset$	$\emptyset$
b	$\emptyset$	$\{(p2, 17)\}$
c	$\{(x, p3)\}$	$\{(p4, 17)\}$ .

The potential alias for procedure  $c$  arises when call site  $a$  passes the global variable  $x$  as an actual parameter. The constants come about from passing

<sup>2</sup> For consistency with the rest of the literature on interprocedural data-flow analysis, we will call this set  $\text{REF}$ , even though we have used the name  $\text{USE}$  in the past.  $\text{USE}$  appears in several sources as the set of variables whose values can be read before modification.  $\text{REF}$  ignores the issue of whether or not a modification intervenes between the call site and the first use in a called procedure. Thus, the  $\text{REF}$  set is inherently flow-insensitive, while the  $\text{USE}$  set is inherently flow-sensitive.

```

program a
integer x,y,z,v1,v2
common /global/x,y,z
...
v2 = 17
α: call c(x, v2)
...
β: call b(v1, v2)
...
v2 = v2 * x
...
end

subroutine b(p1, p2)
integer p1, p2
...
γ: call c(p1, p2)
p2 = p1 * 3
...
end

subroutine c(p3, p4)
integer x,y,z,p3,p4
common /global/x,y,z
...
p3 = p4 * 2
...
end
    
```

Fig. 2. Example program fragment.

	Data-flow problem	Type	Flow type	Flow Direction
Global common subexpressions	AVAIL	I	all-path	forward
Code hoisting	VERYBUSY	I	all-path	backward
Global constant propagation	REACH	I	augmented any-path	forward
Register store elimination	LIVE	II	any-path	backward

Fig. 3. Summary of examples.

the constant valued variable *v2* as an actual at  $\alpha$  and  $\beta$ ;  $\gamma$  simply passes the value through to procedure *c*. The summary sets for the program fragment would be:

Call site	MOD	REF
$\alpha$	{x}	{v2}
$\beta$	{v1, v2}	{v1, v2}
$\gamma$	{p1}	{p2}.

The only statements that either modify or use the value of a variable are the three assignments. The MOD and REF information arises from the assignments in procedures *b* and *c*, along with parameter bindings at the various call sites.

#### 4. THE GENERAL FRAMEWORK

We have formulated our techniques for recompilation analysis as a test that determines when a procedure must be recompiled. All of our techniques apply the same test; they compare the current interprocedural information for a

procedure against previously recorded *annotation sets*. The annotation sets contain those interprocedural facts that can be true without invalidating the procedure's previous compilation. Our specific implementation techniques differ in the precision with which they assign values to these annotation sets. We attach the following sets to the program's call graph:

- (1)  $MayBeAlias(p)$ , for each procedure  $p$ —the set of alias pairs that are allowed without forcing a recompilation. If a change adds a pair to  $ALIAS(p)$  that is not in  $MayBeAlias(p)$ , recompilation is required.
- (2)  $MayMod(e)$ , for each call graph edge  $e$ —the set of variables that may be modified as a side effect of the call without forcing a recompilation. If a change adds a variable to  $MOD(e)$  that is not in  $MayMod(e)$ , recompilation is required.
- (3)  $MayRef(e)$ , for each call graph edge  $e$ —the set of variables that may be used as a side effect of the call without forcing a recompilation. If a change adds a variable to  $REF(e)$  that is not in  $MayRef(e)$ , recompilation is required.
- (4)  $MustBeConstant(p)$ , for each procedure  $p$ —the set of constant pairs that must hold on entry to procedure  $p$  if recompilation is to be avoided. If  $\exists(x, v) \in MustBeConstant(p)$  that is not in  $CONSTANTS(p)$ , recompilation is required.

Given these sets, the recompilation test can be expressed quite simply. A procedure  $p$  must be recompiled if either  $p$  changed or the interprocedural information associated with  $p$  changed and any of the following are true:

- (a)  $ALIAS(p) - MayBeAlias(p) \neq \emptyset$ ,
- (b)  $MOD(e) - MayMod(e) \neq \emptyset$ , for any call site  $e$  in  $p$ ,
- (c)  $REF(e) - MayRef(e) \neq \emptyset$ , for any call site  $e$  in  $p$ , and
- (d)  $MustBeConstant(p) - CONSTANTS(p) \neq \emptyset$ .

Set subtraction is defined so that  $a \in (X-Y)$  if and only if  $a$  is a member of  $X$  and not  $Y$ .

To construct a list of procedures needing recompilation, the recompilation tool first initializes the list to include every procedure where a nontrivial editing change has been made. Trivial changes do not alter the code generated for the procedure; this includes format changes or changes to comments. Next, it updates the program's  $ALIAS$ ,  $MOD$ ,  $REF$ , and  $CONSTANTS$  sets. (Ideally, incremental techniques should be used to update these sets [6, 11, 12].) Whenever this update changes the value of one of these sets, the compiler applies the appropriate test. If the test indicates that recompilation is necessary, the corresponding procedure is added to the recompilation list. Because the analyzer only tests sets that change during the incremental update, the test requires a number of set operations proportional to the size of the region of changed data-flow information.



As an example, consider the following assignment of values to the annotation sets. For each procedure  $p$  let

$$\begin{aligned} \text{MaybeAlias}(p) &= \emptyset \quad \text{and} \\ \text{MustBeConstant}(p) &= \{(x, \Omega), \text{ for all } x \text{ declared in the program}\}, \end{aligned}$$

where  $x$  ranges over the parameters and global variables of  $p$  and  $\Omega$  is a constant value that appears nowhere in the program. For each call site  $e$  let

$$\begin{aligned} \text{MayMod}(e) &= \emptyset \quad \text{and} \\ \text{MayRef}(e) &= \emptyset. \end{aligned}$$

With these annotation sets, the compiler will recompile every procedure where either the source text or some associated interprocedural set has changed. It will not recompile procedures for which the information is unchanged because the test is not applied at those procedures. Hence, this test is a slight improvement over the naive approach of recompiling every procedure.

Consider the impact of deleting the assignment statement from procedure  $b$  in the example program. To determine which procedures must be recompiled, the analyzer begins with  $b$ , the changed procedure. After updating the interprocedural information, it discovers that only two sets have changed:  $\text{MOD}(\beta) = \{v1\}$  and  $\text{REF}(\beta) = \{v2\}$ . Because sets associated with procedure  $a$  have changed, it applies the test to  $a$  and slates it for recompilation. Since none of the sets associated with  $c$  have changed, the analyzer ignores  $c$ . Thus, it determines that only  $a$  and  $b$  must be recompiled.

The effectiveness of the testing procedure used by the recompilation tool depends entirely on the values assigned to *MaybeAlias*, *MayMod*, *MayRef*, and *MustBeConstant*. To improve the precision of the test involves expanding *MaybeAlias*, *MayMod*, and *MayRef* to include more allowed facts, or shrinking *MustBeConstant* to exclude more facts. Three instantiations of this general framework are presented in the following sections. The methods are presented in increasing order of complexity; each successive method gives rise to a recompilation analysis of improved precision. Two simple methods for constructing conservative approximations to precise annotation sets are described in the next two subsections. They rely on information that the compiler produces when computing interprocedural information. Sections 5 and 6 discuss a technique for precisely computing *MaybeAlias*, *MayMod*, *MayRef*, and *MustBeConstant*. This technique is substantially more complex than the other methods described; it requires that the compiler produce annotation sets that indicate which interprocedural facts it relied upon when performing optimizations.

#### 4.1 Most Recent Compilation

Our first approach to computing the annotation sets simply remembers the values of ALIAS, MOD, REF and CONSTANTS used in the most recent compila-

tion of the procedure. In other words, whenever we compile a procedure  $p$ , we set

- (1)  $MayBeAlias(p) = ALIAS_{OLD}(p)$ ,
- (2)  $MayMod(e) = MOD_{OLD}(e)$ , for each call site  $e$  in  $p$ ,
- (3)  $MayRef(e) = REF_{OLD}(e)$ , for each call site  $e$  in  $p$ , and
- (4)  $MustBeConstant(p) = CONSTANTS_{OLD}(p)$ .

These annotation sets yield the following recompilation tests:

- (a)  $ALIAS_{NEW}(p) - ALIAS_{OLD}(p) = \emptyset$ ,
- (b)  $MOD_{NEW}(e) - MOD_{OLD}(e) \neq \emptyset$ , for any call site  $e$  in  $p$ ,
- (c)  $REF_{NEW}(e) - REF_{OLD}(e) \neq \emptyset$ , for any call site  $e$  in  $p$ , and
- (d)  $CONSTANTS_{OLD}(p) - CONSTANTS_{NEW}(p) \neq \emptyset$

where OLD indicates interprocedural information associated with the procedure when it was last compiled and NEW indicates interprocedural information that has been updated for the current compilation of the procedure.

This set of assignments reveals the principles underlying the recompilation tests. The summary and aliasing sets identify events whose occurrence *cannot be ruled out* by the analysis. For example, if a variable is in the MOD set for a given call site, the compiler must assume that it may be modified, but if a variable is absent from the same set, the compiler may safely assume that the value of that variable will be unchanged upon return from the call. In other words, in considering the ALIAS, MOD, and REF sets, the compiler can only depend on what is *not* in the sets. If an optimization is safe when a variable is present in one of these sets, that optimization will still be safe if the variable is removed because the compiler must have already considered the possibility that the associated event might not occur. Hence, changes in these sets necessitate recompilation only when they expand the sets. Thus, a deletion cannot invalidate the correctness of previous compilations, although it can create a new opportunity for optimization. This principle motivates tests (a), (b), and (c).

On the other hand, the  $CONSTANTS(p)$  set contains facts that are true on every path leading to an invocation of  $p$ . Thus, if a pair  $(x, v)$  is in  $CONSTANTS(p)$ , the compiler can rely on  $x$  having value  $v$  on entry to  $p$  and can replace references to  $x$ , on paths where  $x$  is unmodified, with the constant value  $v$ . If a subsequent editing change removes  $(x, v)$  from  $CONSTANTS(p)$ , this forward substitution by the constant value is invalidated. Thus, removing a fact from  $CONSTANTS(p)$  may mandate a recompilation. An addition to  $CONSTANTS(p)$  cannot invalidate a previous compilation, but it can open up new opportunities for optimization. This provides the rationale for test (d).

Consider once again the impact of deleting the assignment statement from procedure  $b$  in our example, assuming that annotation sets have been generated using information from the most recent compilation. The analyzer repeats the steps described earlier, placing  $b$  on the recompilation list because of the editing change and applying the test to procedure  $a$  because of changes to  $MOD(\beta)$  and  $REF(\beta)$ . The test indicates that procedure  $a$  need not

be recompiled, since both of the changes are deletions from flow-insensitive summary sets. Thus, with these annotation sets, the same testing procedure limits the recompilation list to procedure *b*.

#### 4.2 APPEARS Information

Although the direct use of information from the most recent compilation yields a recompilation test that is significantly better than the naive approach, it fails to take full advantage of the information that the compiler could make available. For example, the test from Section 4.1 will recompile a procedure whenever a variable is added to its MOD set, even if that variable does not appear in any executable statement in the procedure. Determining which variables actually appear in the procedure leads immediately to an improved test. The compiler can easily produce the additional information needed to support such a scheme. The sets must be computed anyway as part of the initial information for computing the MOD and REF sets.

To describe the annotation sets for this improved test, we define three additional sets. For a procedure *p*, APPEARS(*p*) is the set of variables either used or modified inside *p*. If the only occurrence of a variable inside *p* is as an actual parameter at some call site in *p*, then the variable need not be included in APPEARS(*p*). APPEARS<sup>+</sup>(*p*) is defined to be the set of all variables either used or modified in *p* or some procedure invoked as a result of executing *p*. Both APPEARS(*p*) and APPEARS<sup>+</sup>(*p*) can be computed trivially from information produced in the summary computation. Finally, the set ALIASAPPEARS(*p*) describes pairs of variables, where one element of the pair appears locally in *p* and the other element appears in *p* or one of the procedures that can be executed as a result of invoking *p*.<sup>3</sup> This set is defined as

$$\text{ALIASAPPEARS}(p) = \{(x, y) \mid x \in \text{APPEARS}(p) \text{ and } y \in \text{APPEARS}^+(p)\}.$$

Given these sets, the annotation sets at compile time can be computed as follows:

- (1)  $\text{MaybeAlias}(p) = \text{ALIAS}_{\text{OLD}}(p) \cup \overline{\text{ALIASAPPEARS}(p)}$ ,
- (2)  $\text{MaybeMod}(e) = \text{MOD}_{\text{OLD}}(e) \cup \overline{\text{APPEARS}(p)}$ , for each *e* in *p*,

<sup>3</sup> Note that when dealing with aliased pairs of variables, it is important to consider aliases that can arise between a variable in *p* and a variable in a procedure *q* invoked either directly or indirectly by *p*. ALIASAPPEARS(*p*) accounts for all such aliases that could arise in future compilations. However, in most compilers that employ interprocedural information, the recompilation test for ALIAS information would be used in conjunction with the recompilation tests for MOD and REF information. In this case, the ALIAS test could simply determine whether or not both members of the alias pair were in APPEARS(*p*) and avoid computing the more complex ALIASAPPEARS(*p*) information. The resulting tests would correctly detect the necessary recompilations because adding an alias between  $x \in \text{APPEARS}(p)$  and  $y \in \text{APPEARS}(q)$  would cause *x* to be added to the MOD and/or REF sets for the call invoking *q* from *p*. The MOD and REF recompilation tests would then determine that *p* should be recompiled.

- (3)  $\overline{\text{MayRef}(e)} = \overline{\text{REF}_{\text{OLD}}(e) \cup \text{APPEARS}(p)}$ , for each  $e$  in  $p$ , and  
 (4)  $\overline{\text{MustBeConstant}(p)} = \{(n, v) \in \text{CONSTANTS}_{\text{OLD}}(p) \mid n \in \text{APPEARS}(p)\}$ .

Substituting these annotation sets into the recompilation tests in Section 4 yields the following recompilation tests:

- (a)  $(\text{ALIAS}_{\text{NEW}}(p) - \text{ALIAS}_{\text{OLD}}(p)) \cap \text{ALIASAPPEARS}(p) \neq \emptyset$ ,  
 (b)  $(\text{MOD}_{\text{NEW}}(e) - \text{MOD}_{\text{OLD}}(e)) \cap \text{APPEARS}(p) \neq \emptyset$ , for any call site  $e$  in  $p$ ,  
 (c)  $(\text{REF}_{\text{NEW}}(e) - \text{REF}_{\text{OLD}}(e)) \cap \text{APPEARS}(p) \neq \emptyset$ , for any call site  $e$  in  $p$ , and  
 (d)  $\{(n, v) \in (\text{CONSTANTS}_{\text{OLD}}(p) - \text{CONSTANTS}_{\text{NEW}}(p)) \mid n \in \text{APPEARS}(p)\} \neq \emptyset$ .

Note that the tests have been refactored to avoid instantiating sets like  $\overline{\text{APPEARS}(p)}$  and  $\overline{\text{ALIASAPPEARS}(p)}$ . These tests are currently employed in the  $\mathbb{R}^n$  programming environment [18]. In practice, they have proven to be both efficient and effective.

Computing the annotation sets from these definitions eliminates spurious recompilations that arise from information about irrelevant variables. In practice, this is important—procedures often contain declarations for global variables that they never reference. FORTRAN codes often contain large COMMON blocks that define many global variables; a given procedure may only use a subset. In other languages, widespread use of include files causes the same phenomenon. In fact, this is one of the phenomena that motivates Tichy and Baker’s work with include files—their system avoids recompiling procedures that rely on a changed include file if the change only involves declarations that are actually irrelevant to the procedure [30].

To see this more graphically, consider adding the statement

$$x = p4 * 17$$

to procedure  $c$  in the example from Figure 2. This changes  $\text{MOD}(\gamma)$  to  $\{p1, x\}$  and  $\text{MOD}(\beta)$  to  $\{v1, v2, x\}$ . Under the most recent compilation test, this would have required recompilation of both  $a$  and  $b$ . Using APPEARS information, the test determines that  $a$  requires recompilation, but  $b$  does not, since  $x$  does not appear in  $b$ .

A word of caution is required at this point. There are optimizations that, on first consideration, appear to be limited to a single procedure but are, in reality, inherently interprocedural. A prime example is converting a sequential loop into a parallel loop. If the loop contains no call sites, the transformation’s scope is limited strictly to the single procedure. If, however, the loop contains a call site, the transformation is really interprocedural in its scope because it changes the dynamic call graph of the program. If the called procedure contains a local, static variable  $x$  with an upwards-exposed use, followed by an assignment to  $x$ , the dependence between the definition of  $x$  in one iteration and the use of  $x$  in the subsequent iteration must be preserved. Burke and Cytron refer to such variables as “hidden variables” [7]. As we have formulated them, the recompilation tests determine when an intraprocedural optimization can be invalidated by a subsequent change in an interprocedural set. If  $x$  is a local, static variable in the called procedure,

$x$  will not appear in the MOD or REF sets for the call site and the recompilation tests will not be able to detect when it is necessary to recompile the calling procedure. Even if  $x$  is a global variable, the APPEARS test will only handle recompilation correctly if  $x$  appears in the calling procedure. Interprocedural optimizations require a more complex treatment; we describe one such method in Section 7.

## 5. COMPILER COOPERATION

The techniques presented in Section 4 compute approximate annotation sets. While the best of these techniques can eliminate many spurious recompilations, the question remains, can we compute more precise annotation sets? Spurious recompilations arise from a simple fact—the compiler cannot capitalize on every interprocedural fact presented to it. Thus, the APPEARS test of Section 4.2 may judge that some change in an interprocedural set mandates a recompilation, even though the fact is actually irrelevant, simply because the compiler had no opportunity to exploit the fact during the previous compilation. This section explores a methodology for computing more precise annotation sets by relying on the compiler to record those interprocedural facts that it actually uses.

To illustrate how such a method would work, consider constructing the *MustBeConstant* sets to be used in determining which procedures need recompilation due to changes in CONSTANTS sets. Understanding how the compiler actually uses CONSTANTS information is crucial. For a procedure  $p$ ,  $\text{CONSTANTS}(p)$  describes facts known to hold on entry to a procedure. The compiler capitalizes on this information by using it to initialize the global constant propagation phase. Information from  $\text{CONSTANTS}(p)$  then percolates into other optimizations from folded constants. During global constant folding, the compiler can easily construct a precise *MustBeConstant* set by adding a pair  $(x, v)$  to *MustBeConstant*( $p$ ) whenever it folds  $v$  into a use of  $x$ .<sup>4</sup>

Computing precise sets for interprocedural summary and aliasing information is not as simple. The compiler can use summary and aliasing information in two very different ways. It can use a fact directly to justify the safety of an intraprocedural optimization at some call site, or it can fold the information into global data-flow information. The results of that global data-flow information can then be used to justify the safety of an intraprocedural optimization. We call this latter kind of use an indirect use. The remainder of this section describes how to compute annotation sets for interprocedural summary and aliasing information that is indirectly used by the compiler. Section

---

<sup>4</sup> The interprocedural constant analysis can also produce sets describing constant values returned by procedures through global variables and call-by-reference formal parameters [9]. Producing exact *MustBeConstant* sets for each call site under such a scheme is more difficult. The optimizer must know which call sites contributed returned values to each folded constant. Obtaining this information requires solving an auxiliary problem similar to those described for AVAIL and VERYBUSY in Section 5.2.

6 discusses constructing annotation sets when the interprocedural summary and aliasing information is used directly.

In order to compute precise annotation sets for indirect uses of interprocedural summary information, the compiler must be able to associate interprocedural information with the intraprocedural data-flow information to which it contributed. This association enables the optimizer to produce a precise list of the interprocedural summary facts actually used to justify the safety of each transformation that it applies. To compute this mapping, the compiler must augment the intraprocedural dataflow information that it computes with lists of the call sites whose summary information contributed to the various sets. Using this map from intraprocedural data-flow facts to call sites, the compiler can construct precise annotation sets. It can determine, at optimization time, precisely which interprocedural facts must be preserved to guarantee the correctness of a particular application of some optimization. Placing these facts in the appropriate annotation sets ensures that the procedure will be recompiled and reoptimized if one of these facts changes.

To demonstrate how this approach would work, we will consider a set of four optimizations and the global data-flow information required to support them. The optimizations are common subexpression elimination, code hoisting, global constant propagation, and eliminating register stores. We will classify each optimization as being one of two types with respect to recompilation analysis. Throughout this discussion, a procedure  $p$  will be represented by its control-flow graph,  $G = (N, E, n_0)$ . The nodes of this graph represent *basic blocks*, sequences of statements with no control flow branches. Procedure invocations can appear in expressions and as part of a sequence of statements in a basic block. The edges  $e = (m, n) \in E$  represent *control flow* between two basic blocks. Control enters the procedure through its entry node  $n_0$ .

To unify the data-flow equations that we are considering, we will use terms similar to those presented in Aho et al. [1]. Consider equations of the form:

$$out[b] = \bigwedge_{a=f(b)} (gen[a] \cup (out[a] \cap nkill[a])) \quad (1)$$

where  $out[b]$  contains the meet over all paths solution for block  $b$ ,  $gen[a]$  contains local information generated in block  $a$ , and  $nkill[a]$  contains those facts not invalidated in block  $a$ . Here,  $\bigwedge$  is the appropriate meet operator,  $\cup$  or  $\cap$ . Finally, let  $P[b]$  be the set of *predecessors* of  $b$  in the flow graph and  $S[b]$  be the set of *successors* of  $b$  in the flow graph. Then,  $f[b]$  is either  $P[b]$  or  $S[b]$ , depending on whether the problem is a forward or backward flow problem. Note that equations of this form correspond to putting the data-flow functions on the edges of the flow graph.

### 5.1 CALLSBETWEEN Sets

To capture the information needed to compute more precise annotation sets, the compiler will need to compute some auxiliary information during its standard global flow analysis. Assume that we have a data-flow fact  $\alpha$  and a

block  $b$  in the control-flow graph where the presence of  $\alpha$  is used to justify an optimization in block  $b$ . Then, to understand the impact that a specific interprocedural fact has on the values of the sets produced by *forward* data-flow analysis, the compiler will need to determine the set of call sites between the last occurrence of data-flow fact  $\alpha$  and block  $b$ , along all paths leading from  $\alpha$  to  $b$ . We will call that set  $\text{CALLSBETWEEN}(\alpha, b)$ . For *backward* data-flow problems, the compiler will need to determine the set of all call sites between block  $b$  and the first occurrence of dataflow fact  $\alpha$ , along all paths leading from  $b$  to  $\alpha$ . We will call that set  $\text{CALLSBETWEEN}(b, \alpha)$ . For the sake of simplicity, we will refer to both types of information as  $\text{CALLSBETWEEN}$  sets and assume that the difference is clear from the order and type of the subscripts. Thus, for our purposes, the direction of the data-flow equation affects the way that  $\text{gen}(a)$  is defined, whether  $f(b)$  is  $P(b)$  or  $S(b)$ , and the region of interest considered for  $\text{CALLSBETWEEN}$ .

To compute  $\text{CALLSBETWEEN}$  sets, we will expand the domain of the equations that define the global data-flow problems used to support the optimization. We refer to this process as solving an auxiliary data-flow problem. In the original formulations, the elements of the various sets, *out*, *gen*, and *nkill*, are names of data-flow facts. Thus, the presence of an element  $\alpha$  in  $\text{out}[b]$  still denotes that  $\alpha$  is a data-flow fact holding at  $b$ , but it is now represented in terms of a pair  $(\alpha.\text{name}, \alpha.\text{calls})$ , where  $\alpha.\text{name}$  represents the literal name of the data-flow fact and  $\alpha.\text{calls}$  is  $\text{CALLSBETWEEN}$ . To solve this auxiliary problem, we expand the domain of the *gen* and *nkill* sets accordingly:

- For  $\alpha \in \text{gen}[b]$ , if the data-flow problem is a *forward* problem,  $\alpha.\text{calls}$  is the set of call sites in  $b$  *after the last* occurrence of  $\alpha$ ; if it is a *backward* problem,  $\alpha.\text{calls}$  is the set of call sites in  $b$  *before the first* occurrence of  $\alpha$ .
- For  $\alpha \in \text{nkill}[b]$ ,  $\alpha.\text{calls}$  is the set of all call sites in  $b$ .

We must also extend the definitions of the operators to work over the expanded domain. The new interpretations are:

$X \cap Y$  To compute  $X \cap Y$ , for each element  $x \in X$  such that  $\exists y \in Y$  with  $x.\text{name} = y.\text{name}$ , add  $(x.\text{name}, x.\text{calls} \cup y.\text{calls})$  to the result. Note that Eq. (1) includes the term  $(\text{out}[a] \cap \text{nkill}[a])$  rather than the standard, equivalent term  $(\text{out}[a] - \text{kill}[a])$  because it is necessary to accumulate call-site information about the  $\text{nkill}[a]$  sets, which do not appear when the term is written in its standard form.

$X \cup Y$  To compute  $X \cup Y$ , first determine the set *Yonly*, containing every element of  $Y$  whose name does not appear in  $X$ . Then the desired result is the natural union of  $X$  and *Yonly*. Note that  $X \cup Y$  as defined here is not a commutative operation. For this scheme to work correctly,  $X$  must represent  $\text{gen}[a]$  and  $Y$  must represent  $(\text{out}[a] \cap \text{nkill}[a])$ . To avoid overloading the meaning of the  $\cup$  operator,  $X \otimes Y$  will represent the natural union of  $X$  and *Yonly* throughout the remainder of this paper.

$X \wedge Y$  To compute  $X \wedge Y$ , if the appropriate meet operation is  $\cup$ , then for each element  $x \in X$  such that  $\exists y \in Y$  with  $x.name = y.name$ , add  $(x.name, x.calls \cup y.calls)$  to the result. For each  $x \in X$  (and  $y \in Y$ ) where there is no  $y \in Y(x \in X)$  with  $x.name = y.name$ , add  $x(y)$  to the result. If the appropriate meet operation is  $\cap$ , perform the intersection as defined for  $X \cap Y$  above. Note that in both cases,  $\alpha.calls$  is computed as the natural union of the call sites from all the appropriate paths.

Once we have reformulated the problem in this manner, we can solve it using traditional global dataflow techniques. The solution to the reformulated problem contains both the solution to the original problem, encoded as the *name* fields of set elements, and the additional CALLSBETWEEN sets, encoded as the *calls* fields of set elements. We will use this technique in each of our four examples.

## 5.2 Type I Optimizations

The type I optimizations rely on the presence of a fact in the set  $out[b]$  to justify the safety of applying the transformation. The three problems that we consider, global common subexpression elimination, code hoisting, and global constant propagation, are each formulated as a data-flow computation followed by selective application of a transformation. The decision to apply the transformation is based on the results of the data-flow analysis.

*5.2.1 Common Subexpression Elimination.* When the compiler discovers two or more instances of a single expression separated by code that does not redefine any of the variables used in the expression, it can save the result of the first evaluation and replace the subsequent evaluations with a simple reference to the saved value. To locate opportunities for this optimization, known as *global common subexpression elimination*, the compiler must know which expressions are *available* at various points in the procedure. An expression is available on entry to a basic block  $b$  if, along every path leading to  $b$ , the expression has been evaluated since the most recent redefinition of its constituent variables [1]. To represent this information, we associate a set  $AVAIL(b)$  with each block  $b$ .  $AVAIL(b)$  contains all expressions available on entry to  $b$ . These sets can be derived by solving a forward data-flow analysis problem. The following system of equations describes the problem:

$$AVAIL(b) = \bigwedge_{a \in P(b)} (DEF(a) \cup (AVAIL(a) \cap NKILL(a)))$$

where  $P(b)$  is the set of predecessors of  $b$ .  $DEF(a)$  contains those expressions computed in  $a$  and not subsequently redefined in  $a$ .  $NKILL(a)$  is the set of expressions *not* redefined in  $a$ . This system of data-flow equations is *rapid* in the sense of Kam and Ullman [21], so it can be solved efficiently using iterative techniques.

Expressions remain available as long as they are included in  $NKILL(b)$ . For a block  $b$ ,  $NKILL(b)$  excludes any expression containing a variable killed



locally in  $b$ . In the absence of summary information about call sites in  $b$ , the AVAIL analysis must assume that a procedure call kills every variable it can access. Thus, if  $b$  contains a call site,  $\text{NKILL}(b)$  must *exclude* all expressions containing actual parameters and global variables that can be modified as a side effect of the call. If summary information is available, this exclusion can be reduced to the set of expressions involving variables contained in  $\text{MOD}(e)$  for the call site  $e$ .  $\text{REF}(e)$  plays no role in the AVAIL computation.

When a variable  $v \in \text{MOD}(e)$ , no expression containing  $v$  can be in  $\text{NKILL}(b)$  for the block  $b$  containing call site  $e$ , because  $v$  may be modified by execution of the procedure call. Thus, if an expression  $\alpha \in \text{AVAIL}(b)$  for some block  $b$ , its constituent variables cannot be in the MOD set of any call site between  $\alpha$ 's most recent evaluation and  $b$ , on each path leading to  $b$ . If the compiler eliminates a reevaluation of  $\alpha$ , the correctness of that decision relies on the values of the MOD sets for the appropriate call sites. The procedure will need to be recompiled if any of the variables used in  $\alpha$  are added to one of these MOD sets.

To capture this information in the annotation sets, the compiler can compute  $\text{CALLSBETWEEN}$  sets along with the AVAIL information and use them to compute  $\text{MayMod}(e)$ . The  $\text{CALLSBETWEEN}$  sets are computed as described in Section 5.1. For each  $\alpha \in \text{AVAIL}(b)$ ,  $\alpha.\text{calls}$  is  $\text{CALLSBETWEEN}(\alpha, b)$ . The following definitions are used for the local sets.

- For  $\alpha \in \text{DEF}(b)$ ,  $\alpha.\text{calls}$  is the set of call sites in  $b$  *after the last* definition of  $\alpha$ .
- For  $\alpha \in \text{NKILL}(b)$ ,  $\alpha.\text{calls}$  is the set of all call sites in  $b$ .

The operations used are those described in Section 5.1. In this case, the meet operator is intersection. Using these definitions, for each  $\alpha \in \text{AVAIL}(b)$ ,  $\alpha.\text{calls}$  corresponds to the set  $\text{CALLSBETWEEN}(\alpha, b)$ . Even with the changes in the operators and local sets in the AVAIL computation, calculation of the new AVAIL and  $\text{CALLSBETWEEN}$  information is still *rapid* in the sense of Kam and Ullman [21].

Given  $\text{CALLSBETWEEN}(\alpha, b)$ ,  $\text{MayMod}(e)$  can be constructed for common subexpression elimination as follows:

- (1) Initially, let  $\text{MayMod}(e) = \text{ALLVARS}$ , the set of all actual parameters and global variables, for each call site  $e$  in  $p$ .
- (2) Whenever an evaluation of an available expression  $\alpha$  is replaced in block  $b$ , the compiler removes all constituent variables of  $\alpha$  from  $\text{MayMod}(e)$ , for each call site  $e$  in  $\text{CALLSBETWEEN}(\alpha, b)$  and each call site  $e$  inside  $b$  occurring before the optimization.<sup>5</sup>

The resulting  $\text{MayMod}$  sets model the recompilation dependences introduced by applying this optimization.

<sup>5</sup>The optimizer has assumed that these variables are not in  $\text{MOD}(e)$  at each of these call sites.

**5.2.2 Code Hoisting.** An expression is *very busy* at a point  $p$  in a program if, along every path leading from  $p$ , the expression is evaluated prior to redefinition of any of its constituent variables. When the compiler discovers that an expression is very busy at  $p$ , it can evaluate the expression at  $p$ , save the result of this evaluation, and replace the subsequent evaluations with a simple reference to the saved value. This transformation, called *code hoisting*, reduces the total code space required for the procedure [4]. To locate opportunities for code hoisting, the compiler must know which expressions are very busy at various points in the procedure. To represent this information, we associate with each block  $b$  a set  $\text{VERYBUSY}(b)$  that contains all expressions that are very busy upon exit from  $b$ .

To find opportunities for code hoisting, the optimizer can compute the sets of *very busy expressions*. These sets can be derived by solving a backward data-flow analysis problem. The following system of equations describes the problem:

$$\text{VERYBUSY}(b) = \bigwedge_{a \in S(b)} (\text{USED}(a) \cup (\text{VERYBUSY}(a) \cap \text{NKILL}(a))).$$

Here,  $\text{USED}(a)$  contains those expressions computed in  $a$  prior to redefinition in  $a$  of any of its constituent variables.  $\text{NKILL}(a)$  is the set of expressions *not* redefined in  $a$ .

When a variable  $v \in \text{MOD}(e)$ , no expressions containing  $v$  can be in  $\text{NKILL}(b)$  for the block  $b$  containing  $e$ . Thus, if an expression  $\alpha \in \text{VERYBUSY}(b)$  for some block  $b$ , its constituent variables cannot be in the  $\text{MOD}$  set of any call site between the end of  $b$  and the first evaluation of  $\alpha$  on each path leading from  $b$ . To apply the hoisting optimization, the compiler would move the evaluation of  $\alpha$  to the end of  $b$ , store the result in a temporary, and replace each of the subsequent evaluations with a reference to the temporary. The correctness of the decision to hoist  $\alpha$  relies on the values of the  $\text{MOD}$  sets for the call sites between  $b$  and each of the replaced evaluations. The procedure will need to be recompiled if any of the variables used in  $\alpha$  are added to one of these  $\text{MOD}$  sets.

To capture this information in the annotation sets, the compiler can compute auxiliary information in the form of  $\text{CALLSBETWEEN}$  sets as described in Section 5.1. For each  $\alpha \in \text{VERYBUSY}(b)$ ,  $\alpha.\text{calls}$  represents the set  $\text{CALLSBETWEEN}(b, \alpha)$ . The local sets for the auxiliary problem are defined as:

- For  $\alpha \in \text{USED}(b)$ ,  $\alpha.\text{calls}$  is the set of call sites in  $b$  before the first definition of  $\alpha$ .
- For  $\alpha \in \text{NKILL}(b)$ ,  $\alpha.\text{calls}$  is the set of all call sites in  $b$ .

The operations used are those described in Section 5.1. The meet operator is intersection. The dataflow problem is still *rapid* in the sense of Kam and Ullman, even after the addition of the auxiliary problem [21].

Given  $\text{CALLSBETWEEN}(b, \alpha)$ ,  $\text{MayMod}(e)$  can be updated for code hoisting in the following manner:

- (1) Initially, let  $\text{MayMod}(e) = \text{ALLVARS}$ , the set of all actual parameters and global variables, for each call site  $e$  in  $p$ .

- (2) Whenever the optimizer moves a very busy expression  $\alpha$  to the end of block  $b$ , the compiler should remove each of the constituent variables of  $\alpha$  from  $MayMod(e)$  for each  $\alpha$  in  $CALLSBETWEEN(b, \alpha)$ .

The resulting  $MayMod$  sets describe the compilation dependences introduced by code hoisting.

**5.2.3 Global Constant Propagation.** In *global constant propagation*, the optimizer replaces an expression with a constant value if the value can be computed at compile time.<sup>6</sup> This optimization is based on *reaching definitions* information. A definition reaches a particular point  $p$  in a program if there exists a path between it and  $p$  along which the defined variable is not redefined. To represent this information, we associate a set  $REACH(b)$  with each basic block  $b$ .  $REACH(b)$  contains all definitions that reach the entry to block  $b$ . These sets can be derived by solving the following forward data-flow problem.

$$REACH(b) = \bigwedge_{a \in P(b)} (DEF(a) \cup (REACH(a) \cap NKILL(a))).$$

Here,  $DEF(a)$  contains those definitions in  $a$  of variables that are not subsequently redefined in  $a$ .  $NKILL(a)$  is the set of definitions for which the defined variable is not redefined in  $a$ .

When constant propagation is performed, a use of a variable  $x$  can be replaced by a constant  $c$  only if all definitions of  $x$  that reach the use have been recognized as having the value  $c$ . For a use of  $x$  that is replaced by  $c$  at a point  $p$ , any call sites that can be executed prior to  $p$  can potentially invalidate the optimization. If  $x$  is subsequently added to the  $MOD$  set of some such call site, that change represents a potential change in  $x$ 's value. In the absence of better interprocedural information, this new definition invalidates the forward substitution of  $c$  for  $x$  at  $p$ .

To account for this interaction between interprocedural  $MOD$  sets and the global  $REACH$  sets, we can compute auxiliary  $CALLSBETWEEN$  sets in the manner described in Section 5.1. For each  $\alpha \in REACH(b)$ ,  $\alpha.calls$  represents the set  $CALLSBETWEEN(\alpha, b)$ . In this case, we use the following definitions for the local sets.

- For  $\alpha \in DEF(b)$ ,  $\alpha.calls$  is the set of call sites in  $b$  *after the last* definition of  $\alpha$ .
- For  $\alpha \in NKILL(b)$ ,  $\alpha.calls$  is the set of all call sites in  $b$ .

The operations used are those described in Section 5.1. The meet operator is set union. With these definitions, the revised  $REACH$  equations will compute both reaches information and the  $CALLSBETWEEN$  sets.<sup>7</sup> The revised computation is still rapid in the sense of Kam and Ullman [21].

<sup>6</sup> Where interprocedural constant propagation is performed, the  $CONSTANTS$  sets are used as initial information in the single procedure, or global, computation.

<sup>7</sup> Note that in this case,  $X \cap Y$  is the empty set, where  $X = DEF(a)$  and  $Y = (REACH(a) \cap NKILL(a))$ . So, we could use  $X \cup Y$  instead of  $X \otimes Y$ . However, it is correct as defined here and it fits the proposed framework.

Given the CALLSBETWEEN sets, we can compute *MayMod* sets that are more precise than those derived using APPEARS information. To update *MayMod*( $e$ ):

- (1) Initially, let  $MayMod(e) = ALLVARS$ , the set of all actual parameters and global variables, for each call site  $e$  in  $p$ .
- (2) Whenever a variable  $x$  is replaced by a constant, the compiler must update the *MayMod* sets for any call site that lies on a path between a definition of  $x$  and the replacement site. These are the call sites in the sets  $CALLSBETWEEN(\alpha, b)$  for each definition  $\alpha$  of  $x$  in  $REACH(b)$ , where  $b$  is the block containing the replacement. Additionally,  $x$  should be removed from *MayMod*( $e$ ) for each call site inside block  $b$  occurring before the replaced reference.<sup>8</sup>

The *MayMod* sets computed this way, however, are still approximate. When an assignment is added in some other procedure, causing  $x$  to appear in the MOD set of some call site  $e$ , we do not know the value that  $x$  receives. It is possible that  $x$  receives the value  $c$  at the new assignment, too. If the interprocedural analysis finds constant values returned by procedures, the *MayMod* sets can be computed in a more precise manner to account for those returned constants [9].

### 5.3 Type II Optimizations

Where type I optimizations depend on the presence of a fact in the set  $out[b]$ , type II optimizations depend on the absence of a fact from  $out[b]$ . As an example, we consider register store elimination, which depends on the absence of a variable from LIVE sets to remove the last store of a value. This changes the CALLSBETWEEN information that we are interested in computing in two important ways.

- (1) The information of interest is associated with facts *not* in the set. In the type I optimizations, it was associated with facts in the set. Thus, we are interested in the  $\alpha.calls$  fields of facts that would correspond to the zeroes in a bit-vector implementation.
- (2) The set  $CALLSBETWEEN(b, \alpha)$  now describes a region between  $b$  and a point at which the optimizer decided that some event involving  $\alpha$  *did not* occur. In the case of register store elimination, if  $\alpha$  is not LIVE at  $b$ , CALLSBETWEEN contains all call sites between block  $b$  and a redefinition of  $\alpha$  (or an exit from the procedure if no redefinition exists) along every path leaving  $b$ .

We would like to use the framework described in Section 5.1 to compute CALLSBETWEEN information for type II optimizations. For this framework to be applicable, we must compute  $out[b]$ , the data-flow information that the compiler actually uses when it performs type II optimizations. Thus, when

<sup>8</sup> If an expression is replaced, rather than a simple variable, the *MayMod* sets (at the same set of call sites) must be updated to remove each of the expression's constituent variables.

the optimizer relies on the absence of a fact from some data-flow set, we recast the problem to compute the complement of that set, so that the transformation can be based on the presence of a fact in the complement. This insures that the CALLSBETWEEN information for the facts that the optimizer uses will be accumulated.

To determine how to transform the data-flow equations associated with a type II optimization into the form needed to compute CALLSBETWEEN information using the method described in Section 5.1, we recast our general formulation, Eq. (1) from Section 5, as follows.

$$out_0[b] = \bigwedge_{a=f(b)} (gen_0[a] \cup (out_0[a] \cap nkill_0[a])).$$

Using DeMorgan's law, we compute the equation for  $\overline{out_0[b]}$ . Note that  $\wedge$  is the complement of  $\bigwedge_0$ , where  $\cup$  and  $\cap$  are considered complements.

$$\overline{out_0[b]} = \bigwedge_{a=f(b)} (\overline{gen_0[a]} \cap (\overline{out_0[a]} \cup \overline{ncill_0[a]})).$$

Distribute the intersections over the union to construct a new equation:

$$\overline{out_0[b]} = \bigwedge_{a=f(b)} ((\overline{gen_0[a]} \cap \overline{ncill_0[a]}) \cup (\overline{out_0[a]} \cap \overline{gen_0[a]})).$$

We can redefine this equation to look like Eq. (1):

$$out[b] = \bigwedge_{a=f(b)} (gen[a] \cup (out[a] \cap nkill[a]))$$

with the following assignments:  $out[b] = \overline{out_0[b]}$ ,  $gen[a] = \overline{gen_0[a]} \cap \overline{ncill_0[a]}$ , and  $nkill[a] = \overline{gen_0[a]}$ . Again, CALLSBETWEEN can be computed as described in Section 5.1. The next subsection shows an example based on the use of LIVE information.

**5.3.1 Eliminating Register Stores.** If the compiler discovers a point where the value of a local variable of a procedure exists in a register and that value cannot be used later in the procedure, it need not store the value back into memory. To perform this optimization, called *eliminating unnecessary stores*, the compiler must recognize the last use of a variable in a procedure.

A variable is *live* at a point in a procedure if there exists a control flow path from that point to some use of the variable and that path contains no assignments to the variable. Live analysis associates a set  $LIVE(b)$  with each block  $b$ .  $LIVE(b)$  contains all the variables that are live upon exit from block  $b$ . LIVE sets can be computed by solving a backward data-flow problem. The following equation is a slightly modified version of the equation given by Aho et al. [1]

$$LIVE(b) = \bigwedge_{a \in S(b)} (IN(a) \cup (LIVE(a) \cap NDEF(a))).$$

Here,  $LIVE(b)$  is the set of variables live immediately after block  $b$ ,  $IN(a)$  is the set of variables whose values may be used in  $a$  prior to any definition of

that variable in  $a$ , and  $\text{NDEF}(a)$  is the set of variables not assigned values in  $a$ .

Without summary information for call sites, the compiler must assume that a call references any variables visible to it. This assumption extends the live ranges of variables, inhibiting the application of register store elimination. Interprocedural REF sets can reduce the set of variables assumed LIVE because of a call site. Because  $\text{MOD}(e)$  says nothing about uses, MOD information is not pertinent to the computation of LIVE information.

Register store optimizations are invalidated when the life of a variable is extended by addition of a variable use after the current last use. Thus, any call sites between the eliminated store and the end of the procedure can potentially invalidate a register store optimization. Assume that the optimizer has eliminated the last store of a variable  $x$ . If a subsequent change to some other procedure adds  $x$  to the REF set of a call site that occurs after the eliminated store, the procedure must be recompiled, since the change possibly makes the eliminated store necessary for correct execution of the program.

To construct *MayRef* sets that reflect this dependence on interprocedural REF information in the LIVE sets, we would like to compute auxiliary CALLSBETWEEN sets in the manner described in Section 5.1. Because this is a type II optimization, computing the auxiliary information is more complex. First, we must reformulate the data-flow equations as described in the previous subsection. We recast the equations in terms of  $\overline{\text{LIVE}}(b)$ . Let  $\text{out}[b] = \overline{\text{LIVE}}(b)$ ,  $\text{gen}[a] = \overline{\text{IN}}(a) \cap \text{NDEF}(a)$ , and  $\text{kill}[a] = \overline{\text{IN}}(a)$ . Let the meet operation be set intersection. Now the general equation we gave in Section 5 can be used to compute  $\overline{\text{LIVE}}$ .

It is interesting to note how similar the  $\overline{\text{LIVE}}$  computation is to the other data-flow equations that we have considered. Given this reformulation, we can derive the necessary CALLSBETWEEN sets as auxiliary information during the  $\overline{\text{LIVE}}$  computation. For each  $\alpha \in \text{out}[b]$ ,  $\alpha.\text{calls}$  represents the set  $\text{CALLSBETWEEN}(b, \alpha)$ . The following definitions work within the general framework described in Section 5.1.

- For  $\alpha \in \text{gen}[b]$ ,  $\alpha.\text{calls}$  is the set of call sites in  $b$  before the first definition of  $\alpha$ .
- For  $\alpha \in \text{kill}[b]$ ,  $\alpha.\text{calls}$  is the set of all call sites in  $b$ .

The operations used are those described in Section 5.1. The meet operator is intersection. After all this manipulation, the final data-flow framework for  $\overline{\text{LIVE}}$  with its auxiliary information remains *rapid* in the sense of Kam and Ullman [21].

To construct a recompilation test that precisely characterizes the use of interprocedural information in the register store optimization, we want to enlarge the *MayRef*( $e$ ) set. Given this set, *MayRef*( $e$ ) can be computed as follows:

- (1) Initially, let  $\text{MayRef}(e) = \text{ALLVARS}$ , the set of all actual parameters and global variables, for each call site  $e$  in  $p$ .

- (2) Whenever a store of a variable  $v$  is eliminated, the optimizer removes  $v$  from  $MayRef(e)$  for each call site  $e$  in  $CALLSBETWEEN(b, \alpha)$  and each call site inside  $b$  occurring after the optimization.

This results in  $MayRef$  sets that precisely capture the recompilation dependencies for this optimization.

#### 5.4 Rationale

In each of the four examples, we were able to construct more precise annotation sets by using  $CALLSBETWEEN$  sets computed as an auxiliary data-flow problem. The  $CALLSBETWEEN$  information associated with a fact follows the path that the fact takes through the procedure during the data-flow computation. When a fact is generated in a basic block, the set  $\alpha.calls$  associated with it takes into account the call sites between the point where it was generated and that end of the block where the data-flow computation exits. In a backward flow computation, this is the beginning of the block; for a forward flow computation, it is the end of the block. When a fact  $\alpha$  passes through a block unchanged, all of the call sites in the block are added to  $\alpha.calls$  because the block is an  $\alpha$ -clear path.

The operations used in solving the auxiliary data-flow problems are straightforward. Whenever multiple paths come together, some data-flow facts are invalidated and some are not. Any fact that is not invalidated has an  $\alpha.calls$  set that contains the natural union of  $\alpha.calls$  sets from the individual facts that made the new fact valid.

The only operator that is unusual is the  $X \otimes Y$  operator. The reason for this operator is somewhat subtle. The standard data-flow equations use binary information. In extending the underlying data-flow equations to correctly compute  $CALLSBETWEEN$  sets, we expanded each bit in the original bit-vector to include both the bit and a set that we designate that bit's  $calls$  set. We designate the original bit as the fact's *name*. The set operations on these expanded objects are based on the presence or absence of the name, i.e., the value of its original bit. In this framework, the result of  $X \cup Y$  and  $X \otimes Y$  are not the same. (Recall the definition in Section 5.1.) Furthermore, it is clear that the data-flow events that are of interest to an optimizer are those computed with the  $X \otimes Y$  operation because these are the events that happened nearest to the point of optimization. For example, consider a basic block  $b$  that contains two computations of expression  $e$ . The terms  $e_1$  and  $e_2$  refer to the first and the second instantiation of  $e$ , respectively. Assume that  $e \in AVAIL(b) \cap NKILL(b)$ . If the optimizer applies common subexpression elimination to  $e_2$ , the optimization is safe as long as none of the variables in  $e$  are redefined between  $e_1$  and  $e_2$ . The fact that  $e$  was also available on all paths leading to  $b$  is immaterial because all of the paths to  $e_2$  pass through  $e_1$ . Hence, only the call sites between  $e_1$  and  $e_2$  need to be recorded to insure safety. Stated in more general terms, for a fact in both  $X$  and  $Y$ , its presence in  $Y$  is irrelevant because the occurrence in  $X$  happens on the path to the occurrence in  $Y$ . If the operators in the standard data-flow equations were more descriptive, we could merely state that we compute the  $\alpha.calls$  set for a

fact at any point by doing a natural union of the  $\alpha.calls$  sets from all of the paths contributing to that fact.

The table in Figure 3 summarizes the data-flow information used in our example optimizations. Global common subexpression elimination and code hoisting clearly depend on all-paths information. The AVAIL and VERYBUSY information that we compute for these optimizations is all-paths information. Even optimizations that, at first glance, appear to depend on any-path information actually depend on all-paths information. Global constant propagation uses REACH information. REACH is any-path information, but global constant propagation depends on an augmented form of this information. It computes all-paths information—the definition reaches point  $p$  with known constant value  $c$ . For a constant  $c$  to be folded at  $p$ , the same constant value  $c$  must reach  $p$  along all paths through the procedure leading to  $p$ . Finally, register store elimination is usually based on any-path LIVE information. However, the information that is actually used by this optimization is the all-paths LIVE information discussed in Section 5.3. This is true of other optimizations, like dead-code elimination, that are based on the converse of any-paths information.

Our examples illustrate that optimizations based on global data-flow information are either based on all-paths information (type I) or the converse of any-paths information (type II). In either case, the information actually used by the optimization is all-paths information. This follows from the simple observation that, along all paths through the program, the optimization must preserve the program’s semantics. Thus, the correctness of the optimization is based on the behavior of the program along all paths (and, therefore, on the meet-over-all-paths solution for some data-flow problem) [26, 29].

The information that we compute for CALLSBETWEEN is any-path information because optimizations are based on all-paths information. That is, if an event along any path to the optimization is invalidated, the optimization itself is invalidated because it relied on all-paths information. The any-path information that we compute for recompilation analysis leads to a precise test for recompilation due to changes in interprocedural information because it allows us to detect if any path between an event and an optimization that depends upon that event is broken. Since optimizations rely on the fact that none of these paths are broken, we know that recompilation is necessary if any of the paths are broken.<sup>9</sup>

### 5.5 Complexity

Adding the computation for CALLSBETWEEN information to the global data-flow analysis phase increases the time and space that are required to compute the

<sup>9</sup> Unless, of course, the editing change to the program makes no real difference in the values being passed around. Consider adding an assignment to some procedure that enlarges the MOD set but does not change the values of any variable on return from the procedure. If we assign a variable its known constant value, we really do not invalidate the application of a constant fold, but the MOD-based test will dictate recompilation. This is another example of the limit of precision—the analog of the “up to symbolic evaluation” condition that Barth gave for summary information.



global data-flow information by a factor of  $O(p)$  where  $p$  is the number of call sites in the procedure. The additional space is used to store, with every data-flow fact in every basic block in the program, the set of call sites associated with that fact. If the set of call sites is stored as a bit vector, each set requires a bit vector of length  $p$ . In effect, we have a  $k$  by  $p$  bit matrix, where  $k$  is the number of data-flow facts.

Additional time is needed to update the set of call sites associated with the data-flow facts. To update the call sites information during the data-flow computation, we compute, for each call site in the bit matrix, those facts that rely on interprocedural information provided by that call site. This computation requires a constant number of bit vector operations on bit vectors of length  $k$  for each of the  $p$  call sites in the procedure. Hence, the time required to compute global data-flow information is  $O(pEd(G))$  for reducible graphs and  $O(pEN)$  for nonreducible graphs, where  $E$  is the number of edges,  $N$  is the number of basic blocks in the flow graph of the procedure, and  $d(G)$  is the loop-connectedness of the graph as defined by Kam and Ullman [21].

Since nested procedures occur inside a single compilation unit, an optimization that saves both time and space is possible. A clever implementation can capitalize on the fact that variables not visible to the calling procedure need not be represented in the CALLSBETWEEN set. This is safe because changing the visibility of variables inside a procedure requires an editing change—an act that mandates its recompilation.

## 5.6 Generalization

Examining our four sample optimizations leads to the following general algorithm for constructing precise annotation sets. The compiler assigns the annotation sets values that would *never* mandate recompilation and then adjusts the sets to reflect each transformation, as applied. The sets get the following initial values:

- (1)  $MayBeAlias(p) = ALLVARS \times ALLVARS$ ,
- (2)  $MayMod(e) = ALLVARS$ , for each call site  $e$  in  $p$ ,
- (3)  $MayRef(e) = ALLVARS$ , for each call site  $e$  in  $p$ , and
- (4)  $MustBeConstant(p) = \emptyset$ .

Whenever an interprocedural fact is used to justify the safety of an optimization, the appropriate set is adjusted, subtracting from  $MayBeAlias$ ,  $MayMod$ , or  $MayRef$ , or adding to  $MustBeConstant$ .

The construction of  $MustBeConstant$  was described in Section 5. By considering the computation of  $MayMod$  and  $MayRef$  for the four example optimizations, we can develop a general strategy toward computing  $MayMod$  and  $MayRef$  sets with respect to optimizations based on global data-flow information.

We distinguish between two respects in which an addition to MOD can change global data-flow information. First, it contributes a new definition that reaches certain points in the program. This adds definitions to REACH

sets and can affect all-paths information that is related to REACH information. Our discussion of updating *MayMod* sets for global constant propagation illustrates the general strategy for accommodating this kind of impact. Second, it can affect the reaching, exposure, and availability characteristics of other definitions, uses, and expressions, respectively (i.e., it can kill them). In the same manner that MOD definitions preserve the REACH characteristics of other definitions, they preserve any-path global data-flow information in general. Thus, this latter impact is only important with respect to all-paths information. Our discussion of updating *MayMod* sets for common sub-expression elimination and code hoisting illustrates the accommodation of this kind of impact.

As with MOD and REF information, ALIAS information is factored into global data-flow information. To determine when a procedure  $p$  needs to be recompiled due to changes in  $\text{ALIAS}(p)$ , it is necessary to track which facts the compiler indirectly used when performing optimizations. This can be accomplished by annotating data-flow facts in a manner analogous to the one used for MOD and REF information.

When a new pair is added to  $\text{ALIAS}(p)$ , definition points for one member of the alias pair become definition points for the other member of the pair. Likewise, uses of one member of the alias pair become uses of the other member of the pair. Because of this, adding a new pair to  $\text{ALIAS}(p)$  can invalidate optimizations by effectively adding definitions and uses to the routine. Optimizations based on the type of data-flow information described in Section 5.2 can be invalidated by adding definitions; optimizations based on the type of data-flow information described in Section 5.3 can be invalidated by adding uses.

To precisely determine if an alias pair should not appear in  $\text{MayBeAlias}(p)$ , the compiler computes either  $\text{DEFINEDBETWEEN}$  or  $\text{USEDBETWEEN}$  information for each data-flow fact that involves either a global variable or a formal parameter.  $\text{DEFINEDBETWEEN}(\alpha, b)$  contains those global variables and formal parameters defined on paths contributing to the correctness of data-flow fact  $\alpha$  in basic block  $b$ ;  $\text{DEFINEDBETWEEN}$  is computed when the data-flow information described in Section 5.2 is employed.  $\text{USEDBETWEEN}(\alpha, b)$  contains those global variables and formal parameters used on paths contributing to the correctness of data-flow fact  $\alpha$  in basic block  $b$ ;  $\text{USEDBETWEEN}$  is computed when the data-flow information described in Section 5.3 is employed. Given the  $\text{DEFINEDBETWEEN}$  or  $\text{USEDBETWEEN}$  information for a particular data-flow fact, the compiler can determine whether or not an alias pair can safely appear in  $\text{MayBeAlias}(p)$  as follows:

- (1) Initially, let  $\text{MayBeAlias}(p) = \text{ALLVARS} \times \text{ALLVARS}$ , the set of all pairs of global variables and formal parameters in  $p$ .
- (2) Whenever the compiler uses data-flow fact  $\alpha$  in block  $b$ , it must remove from  $\text{MayBeAlias}(p)$  all alias pairs consisting of a global variable or formal parameter referenced in  $\alpha$  and a variable in either  $\text{DEFINEDBETWEEN}(\alpha, b)$  or  $\text{USEDBETWEEN}(\alpha, b)$ .

Since the paths that lead to the correctness of  $\alpha$  are the same as the paths involved in the  $\text{CALLSBETWEEN}(\alpha, b)$  computation, the approach used for computing  $\text{CALLSBETWEEN}$  information can be used to compute  $\text{DEFINEDBETWEEN}$  and  $\text{USEDBETWEEN}$  information. The definitions for the operators presented in Section 5.1 are used for the computation;  $\text{gen}[b]$  and  $\text{kill}[b]$  are computed as described for  $\text{CALLSBETWEEN}$  information, except that definitions and uses of global variables and formal parameters are tracked instead of call sites.

This section showed an approach for computing more precise recompilation information for changes in  $\text{CONSTANTS}$ ,  $\text{MOD}$ ,  $\text{REF}$ , and  $\text{ALIAS}$  sets. Computing annotation sets that actually reflect compile-time decisions will probably increase the compile times for individual modules. However, it will also make it possible for the compiler to more precisely determine what needs to be recompiled when an editing change is made—potentially leading to a reduction in the number of procedures that need to be recompiled. Under a scenario where either editing changes are expected to be frequent or compilation is expensive due to the aggressive analysis and optimization employed by the compiler, the additional compile time cost associated with using precise recompilation analysis may be offset by the reduction in the number of recompilations.

## 6. DIRECT USE OF INTERPROCEDURAL FACTS

So far, our discussion has concentrated on finding the recompilation dependences that arise from the contribution of interprocedural data-flow information to global data-flow information. Once interprocedural information is made available to the compiler, it is reasonable to expect that the optimizer will make direct use of the facts where appropriate. To preserve correctness in compiled code, our methods of computing annotation sets must account for such direct use.

As an example, consider the code that gets generated for a procedure call in a language with call-by-reference parameter passing. For simplicity, assume that all registers that are preserved across the call are saved in the calling routine. If the compiler ambitiously keeps values in registers, then it is likely that one or more of the actual parameters at the call site will not have a current copy in storage—that is, in memory rather than in a register.<sup>10</sup> Thus, before the call, the compiler must generate code to store each of the actual parameters and global variables for which the store is not current. Similarly,

<sup>10</sup> The optimizing compiler for  $\mathbb{R}^n$  tries to keep all scalar values in registers. Nonaliased global scalars are assumed to have a correct and consistent storage representation only at call sites and procedure entry and exit. A local scalar  $v$  is assumed to have a storage representation only in the neighborhood of a call where it is passed as an actual parameter. It is stored immediately prior to the call and restored afterward. The other mechanism by which a local scalar variable gets moved from a register to storage is when the register allocator decides that it must spill the variable.

after the call, it may need to refresh the register copies of such values from the store, to ensure that they are current.

If the optimizer has interprocedural MOD and REF sets for the call site, it can do better. Any parameter or global variable that is in a register before the call site and is not contained in the set  $(\text{MOD}(e) \cup \text{REF}(e))$  need not be stored before the call. Thus, the compiler need not generate either the address computation or the store instruction. Similarly, any parameter that is not contained in  $\text{MOD}(e)$  need not be refreshed after the call, allowing the compiler to eliminate both the address computation and the load instruction.

The APPEARS test presented in Section 4.2 will correctly model the recompilation dependences introduced by such optimizations. In fact, eliminating stores before the call has the effect of making the APPEARS test for MOD and REF information precise for global variables. If a global variable that appears in the calling procedure is added to either the MOD or REF set at some call site, recompilation will be needed to insert the store for that parameter before the call site. Otherwise, either a reference inside the called procedure or the restore after the call can receive an incorrect value.

If a more precise annotation set is being computed, in the manner described in Section 5, the compiler will need to record such direct use of facts in the appropriate annotation sets. Thus, for each store eliminated before the call site  $e$ , the compiler would need to remove the variable from  $\text{MayMod}(e)$  and  $\text{MayRef}(e)$ . Similarly, for each refresh eliminated after  $e$ , it would need to remove the variable from  $\text{MayMod}(e)$ .

When the compiler directly uses ALIAS information in its compilation, it is difficult to produce precise recompilation information. This is due to the manner in which the compiler employs ALIAS information. When two variables are potential aliases, the compiler must preserve the relative ordering of their loads and stores. Doing this requires either that the compiler track, pairwise, all uses and definitions of each alias pair, or that it simply treat potential aliases extremely conservatively. Because of the expense and complication involved in the former approach, all compilers with which we are familiar adopt the latter strategy. Since the compiler does not track situations where it reorders loads and stores for variables that are not potential aliases, it is difficult to determine when the compiler has relied upon the absence of a particular alias pair from  $\text{ALIAS}(p)$ . This information is necessary for computing precise  $\text{MayBeAlias}(p)$  sets. Thus, when ALIAS information is used directly in a compiler, efficiency may dictate that the APPEARS test discussed in Section 4.2 be employed.

## 7. LARGER COMPILATION UNITS AND INTERPROCEDURAL OPTIMIZATIONS

Our compilation model assumes that each procedure is a distinct compilation unit. Many compilers treat multiple procedures as an indivisible compilation unit, producing a single object file for all the procedures in the unit. The presence of multiple procedures in a single unit slightly complicates the

recompilation analysis. When analyzing a unit that contains multiple procedures, the compiler must recognize that the procedures are related.

To handle this situation, the compiler can build a pair of maps: one from procedures into compilation units and the other from compilation units into procedures. Using these maps, the analyzer can mark all of the procedures in a unit for recompilation whenever any of its constituent procedures needs recompilation. This can decrease the total amount of analysis required, since it need not test any procedures in a unit already marked for recompilation.

This mechanism also provides a natural way of handling interprocedural optimizations. For our purposes, an interprocedural optimization is an optimization that

- (1) moves code across a call site,
- (2) changes the program's static call graph, or
- (3) changes the program's dynamic call graph.

Examples of these are inline substitution, procedure cloning, and parallelizing a loop containing a call site, respectively.

Clearly, such transformations introduce new compilation dependences between the involved procedures. We can use the maps required for multiple procedure compilation units to take account of such transformations in our testing procedure. The idea is simple; whenever the compiler applies an interprocedural optimization to a pair of procedures that belong to distinct compilation units, these units are treated as if they were a single unit. This strategy requires a straightforward adjustment to each of the two maps described above.

To apply the recompilation test, the analyzer follows the algorithm sketched in Section 4. First, it marks each procedure that has been changed by editing, along with all procedures belonging to the same unit. Next, it updates all of the interprocedural sets. Then, it applies the recompilation test to each procedure where an interprocedural set has changed. Of course, if the procedure is already marked for recompilation, the analyzer need not apply the test. If the test indicates recompilation, the procedure is marked, along with every procedure indicated by the entries in the procedure-to-unit map.

The maps represent the presence of multiple procedures in a compilation unit and express the compilation dependences introduced by interprocedural optimizations. They ensure that the test behaves correctly and efficiently. Each procedure is analyzed independently. When the tests indicate that some procedure must be recompiled, the analyzer marks all procedures in the unit for recompilation. Using the maps can decrease the number of test applications that the analyzer must make.

It is important to recognize the difference between this approach and a hierarchical approach like that found in structural data-flow algorithms. Our approach maintains separate data-flow information for each of the procedures, but accounts for the textual relationships between them. A hierarchical test would merge graph nodes in some structured way. Merging the nodes

for the procedure would simplify the graph, but would result in merging the information used in the recompilation test and losing some precision in the test information. A fact allowed on entry to one procedure might be disallowed on entry to another; if the procedures are both represented by a single node and a single annotation set, the test must indicate recompilation when the fact is added to either path.

## 8. IMPROVED OPTIMIZATION

We have seen that changes in interprocedural information can invalidate the safety of optimizations applied in previous compilation. For the MOD, REF, and ALIAS sets, adding facts to a set associated with a procedure possibly mandated recompiling it, while deleting facts did not. Deletions can, however, open up new possibilities for applying optimizations. Recall that optimizations based on MOD, REF, or ALIAS information rely on the absence of a fact from the data-flow set rather than its presence. Similarly, adding a  $(name, value)$  pair to a procedure's CONSTANTS set can open up opportunities for new optimizations based on knowledge of the constant value.

As stated, our recompilation tests detect when a procedure must be recompiled to ensure consistency with the program in which it will execute. They do not address the issue of detecting potential improvements, although analogous tests can be constructed. For each correctness test in the general framework, a dual test that detects opportunities for improved optimization can be constructed. We introduce four annotation sets for the improvement test: *WereMod*, *WereRef*, *WereAliased*, and *WereConstant*. For each new annotation set, we can formulate a test to predict when recompilation may lead to improved optimization:

- (a)  $WereAliased(p) - ALIAS_{NEW}(p) \neq \emptyset$ ,
- (b)  $WereMod(e) - MOD_{NEW}(e) \neq \emptyset$ , for any call site  $e$  in  $p$ ,
- (c)  $WereRef(e) - REF_{NEW}(e) \neq \emptyset$ , for any call site  $e$  in  $p$ , and
- (d)  $CONSTANTS_{NEW}(p) - WereConstant(p) \neq \emptyset$ .

Again, set subtraction is defined so that  $a \in (X - Y)$  if and only if  $a$  is a member of  $X$  and not  $Y$ . The next subsection describes one possible formulation of these annotation sets and shows how to compute the resulting recompilation tests.

### 8.1 Defining the Annotation Sets

In Section 4.2, we described the approximate annotation sets for the correctness test based purely on static information. Approximate annotation sets for the improvement test can be defined in a similar manner. At each compilation of a procedure  $p$ , the compiler can compute the information used to construct the four annotation sets, based on the interprocedural data-flow sets described in Section 3 and the APPEARS sets described in Section 4.2. Specifically, the sets can be described as:

- (1)  $WereAliased(p) = ALIAS_{OLD}(p) \cap ALIAS_{APPEARS}(p)$ ,
- (2)  $WereMod(e) = MOD_{OLD}(e) \cap APPEARS(p)$ , for each call site  $e$  in  $p$ ,

- (3)  $WereRef(e) = \overline{REF_{OLD}(e) \cap APPEARS(p)}$ , for each call site  $e$  in  $p$ , and  
 (4)  $WereConstant(p) = \{(n, v) \notin \overline{CONSTANTS_{OLD}(p)} \mid n \in APPEARS(p)\}$ .

The rationale for these assignments is analogous to that underlying the correctness test in Section 4.2. Substituting these annotation sets into the recompilation tests in Section 8 and refactoring the equations yields the following recompilation tests:

- (a)  $(\overline{ALIAS_{OLD}(p)} - \overline{ALIAS_{NEW}(p)}) \cap \overline{ALIASAPPEARS(p)} \neq \emptyset$ ,  
 (b)  $(\overline{MOD_{OLD}(e)} - \overline{MOD_{NEW}(e)}) \cap \overline{APPEARS(p)} \neq \emptyset$ , for any call site  $e$  in  $p$ ,  
 (c)  $(\overline{REF_{OLD}(e)} - \overline{REF_{NEW}(e)}) \cap \overline{APPEARS(p)} \neq \emptyset$ , for any call site  $e$  in  $p$ , and  
 (d)  $\{(n, v) \in (\overline{CONSTANTS_{NEW}(p)} - \overline{CONSTANTS_{OLD}(p)}) \mid n \in \overline{APPEARS(p)}\} \neq \emptyset$ .

Note that refactoring eliminates the need to instantiate  $WereConstant(p)$ .

It does not seem reasonable to examine techniques for constructing more precise versions of these sets. That would require the compiler to consider each interprocedural fact and determine whether or not there existed an optimizing transformation that the fact prevented. We believe that this type of analysis would be both difficult to implement and expensive to execute.

## 8.2 Practical Application

Whenever recompilation analysis indicates that a procedure must be recompiled for correctness, the compilation system will recompile it. Unfortunately, deciding to recompile for better optimization is not as simple a choice. First, the compiler may not be able to capitalize on the changed interprocedural sets—the optimization might have been prevented by facts other than the one just changed. Second, even if the optimization can be done, the run-time improvement obtained may not justify the cost of recompilation, particularly if the procedure is large. On the other hand, the changed information might make a major difference—for example, if it exposed a substantial amount of parallelism.

Before we can construct a practical compiler that capitalizes on tests for improved optimization, we need reasonable estimators that can predict run-time improvement as a function of changes to interprocedural facts. Until such an estimator is available, recompiling for improvement is almost certainly a hit-or-miss proposition. The tests that we have presented in this section can be used to tell the compiler which procedures are candidates for such analysis, but they cannot, by themselves, predict the results of recompiling.

## 9. SUMMARY AND CONCLUSIONS

Compiling a program in the presence of interprocedural information introduces dependences between its procedures that complicate the question of what to recompile when a change is made in the program. In the absence of information about these dependences, all procedures in the program must be recompiled whenever a change is made to any one of them. This paper describes a general framework, based upon *annotation sets*, for reducing the

number of unnecessary recompilations required after a change. Within this framework, several methods for computing the annotation sets have been presented. These methods differ in the amount of work required and the precision of the resulting recompilation analysis. The fundamental tradeoff to be evaluated is compilation time versus number of spurious recompilations.

#### ACKNOWLEDGMENTS

Keith Cooper and Ken Kennedy contributed to the initial work in this area and encouraged and supported the extensions presented in this paper. Frances Allen, Ron Cytron, and David Shields of IBM contributed to our discussions of this problem. Barbara Ryder pointed out several problems with the version of this work that was presented at SIGPLAN '86. The TOPLAS referees provided comments and suggestions that improved the exposition of the results presented in this paper. Both the  $\mathbb{R}^n$  and PTRAN implementation teams have provided marvelous research vehicles for experimenting with new ideas about interprocedural analysis and optimization. To all of these people go our heartfelt thanks.

#### REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1986.
2. ALLEN, F. E., BURKE, M., CHARLES, P., CYTRON, R., AND FERRANTE, J. An overview of the PTRAN analysis system for multiprocessing. *J. Parallel Distrib. Comput.* 5, 5 (Oct. 1988), 617-640.
3. ALLEN, F. E., CARTER, J. L., FABRI, J., FERRANTE, J., HARRISON, W. H., LOEWNER, P. G., AND TREVILLYAN, L. H. The experimental compiling system. *IBM J. Res. Dev.* 24, 6 (Nov. 1980), 695-715.
4. ALLEN, F. E., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, J. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1972.
5. AMERICAN NATIONAL STANDARDS INSTITUTE. *American National Standard Programming Language FORTRAN, X3.9-1978*.
- 5A. BANNING, J. An efficient way to find side effects of procedure calls and aliases of variables. In the *Conference Record of the 6th POPL* (San Antonio, Tex., Jan. 1979), 29-41.
6. BURKE, M. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 341-395.
7. BURKE, M., AND CYTRON, R. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. *SIGPLAN Not.* 21, 7 (July 1986), 162-175.
8. CALLAHAN, D. The program summary graph and flow sensitive interprocedural data flow analysis. In *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation*. *SIGPLAN Not.* 23, 7 (July 1988), 47-56.
9. CALLAHAN, D., COOPER, K. D., KENNEDY, K., AND TORCZON, L. Interprocedural constant propagation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. *SIGPLAN Not.* 21, 7 (July 1986), 152-161.
10. CARLE, A., COOPER, K. D., HOOD, R. T., KENNEDY, K., TORCZON, L., AND WARREN, S. K. A practical environment for scientific programming. *IEEE Computer* 20, 11 (Nov. 1987), 75-89.
11. CARROLL, M. Data-flow update via dominator and attribute updates. Ph.D. thesis, Computer Science Dept., Rutgers Univ., New Brunswick, N.J., May 1988.



12. CARROLL, M., AND RYDER, B. G. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT / SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. *SIGPLAN Not.* 22, 1 (Jan. 1987), 171–179.
13. CHOW, F. C. Minimizing register usage penalty at procedure calls. In *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation*. *SIGPLAN Not.* 23, 7 (July 1988), 85–94.
14. COOPER, K., D. Analyzing aliases of reference formal parameters. In *Proceedings of the Twelfth POPL* (New Orleans, La., Jan. 1985), 281–290.
15. COOPER, K. D., AND KENNEDY, K. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*. *SIGPLAN Not.* 23, 7 (July 1988), 57–66.
16. COOPER, K. D., AND KENNEDY, K. Fast interprocedural alias analysis. In *Proceedings of the Sixteenth POPL* (Austin, Tex., Jan. 1989), 49–59.
17. COOPER, K. D., KENNEDY, K., AND TORCZON, L. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. *SIGPLAN Not.* 21, 7 (July 1986), 58–67.
18. COOPER, K. D., KENNEDY, K., AND TORCZON, L. The impact of interprocedural analysis and optimization in the  $\mathbb{R}^n$  programming environment. *ACM Trans. Program. Lang. Syst.* 8, 4 (Oct. 1986), 491–523.
19. DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. *LINPACK Users' Guide*. SIAM, Philadelphia, Pa., 1979.
20. FELDMAN, S. Make—a computer program for maintaining computer programs. *Softw. Pract. Exper.* 9, 4 (1979), 255–265.
21. KAM, J., AND ULLMAN, J. D. Global data flow analysis and iterative algorithms. *J. ACM* 23, 1 (Jan. 1976), 158–171.
22. KAM, J., AND ULLMAN, J. D. Monotone data flow analysis frameworks. *Acta Inf.* 7, 3 (1977), 305–317.
23. MYERS, E. W. A precise interprocedural data flow algorithm. In *Proceedings of the Eighth POPL* (Williamsburg, Va., Jan., 1981), 219–230.
24. RICHARDSON, S., AND GANAPATHI, M. Code optimization across procedures. *IEEE Computer* 22, 2 (Feb. 1989), 42–50.
25. RICHARDSON, S., AND GANAPATHI, M. Interprocedural analysis versus procedure integration. *Inf. Process. Lett.* 32, 3 (Aug. 1989), 137–142.
26. ROSEN, B. K. Monoids for rapid data flow analysis. *SIAM J. Comput.* 9, 1 (Feb. 1980), 159–196.
27. SCHWANKE, R. W., AND KAISER, G. E. Smarter recompilation. Technical Correspondence. *ACM Trans. Program. Lang. Syst.* 10, 4 (Oct. 1988), 627–632.
28. SPILLMAN, T. C. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of IFIP Congress 71* (Ljubljana, Yugoslavia, Aug. 1971), North-Holland, Amsterdam, 376–381.
29. TARJAN, R. E. A unified approach to path problems. *J. ACM* 28, 3 (July 1981), 557–593.
30. TICHY, W. F., AND BAKER, M. C. Smart recompilation. In *Proceedings of the Twelfth POPL* (New Orleans, La., Jan. 1985), 236–244.
31. TICHY, W. F. Smart recompilation. *ACM Trans. Program. Lang. Syst.* 8, 3 (July 1986), 273–291.
32. TORCZON, L. Compilation dependences in an ambitious optimizing compiler. Ph.D. dissertation, Dept. of Computer Science, Rice Univ., Houston, Tex., May 1985.
33. WALL, D. W. Register windows versus register allocation. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*. *SIGPLAN Not.* 23, 7, (July 1988), 67–78.
34. WEGMAN, M., AND ZADECK, F. K. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210.

Received August 1990; revised April 1992; accepted July 1992.