Interprocedural Optimization: Eliminating Unnecessary Recompilation

Keith D. Cooper
Ken Kennedy
Linda Torczon

Department of Computer Science†
Rice University
Houston, Texas

Abstract

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 must develop 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.

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. Substantially improving the efficiency of code generated for a whole program will require a compiler that looks across procedure boundaries during the optimization process. Conradi has estimated that aggressively pursuing such interprocedural analysis and optimization can achieve a twenty percent improvement in run-time speed [Conr 83].

To achieve this goal, a compiler must have information about the inherited naming environment in which a procedure will execute. It must understand the side effects of procedure calls made from within the procedure. It must be free to perform optimizations that span procedure boundaries, like inline substitution, procedure cloning, and interprocedural constant folding [CoKT 86]. In short, it must optimize the whole program. Unfortunately, this ambitious style of interprocedural optimization directly conflicts with separate compilation.

As soon as the compiler uses information about other procedures to make compile-time decisions, the resulting object code becomes a function of the state of the source code for the *entire* program. Later changes to the source text for one procedure can invalidate the correctness of previous compilations of *other* procedures. This happens when an editing session changes the interprocedural characteristics of a procedure and the interprocedural analysis propagates the changed information across the program's call graph. If the previous compilation relied on the old information, some of the compile-time decisions may be invalid in the context of the new interprocedural information. To produce code that is consistent with the new information will require

recompiling.

After a change to a program's interprocedural information, we would like to recompile only procedures with previous compilations that have been invalidated by the change. The goal of *recompilation analysis* is to find the compilation dependences in a program, compare them with changes in the program's interprocedural information, and produce a list of procedures requiring recompilation. The naive approach to solving this problem is to require recompilation of the entire program in response to any change in its interprocedural information. Because this results in recomputing interprocedural information for the whole program at each compilation, it leads to the creation of executables that are both fully optimized and fully consistent with the source code. Although this approach is safe, it is impractical for any production compiler, since it completely discards the economies of separate compilation.

This paper presents 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 manner in which they can be used in an optimizer. The three implementation techniques produce recompilation tests of successively increasing precision, with a concomitant increase in the expense of the test. Any of these three techniques represents a significant improvement over the naive approach.

This problem has not received much attention in the literature, primarily because few compilers have actually computed and used interprocedural information. For example, the PL/I Optimizing Compiler trivializes the problem by limiting its analysis to a single compilation unit [Spil 71]. Other systems, like the ECS project at IBM Research appear to recompile the entire program in response to a single editing change [AlCa 80]. Burke, Cooper, and Kennedy examined this problem briefly. Their collaboration produced an approach similar to the precise analysis for global common subexpression elimination described in Section 3.3.1. They recognized the necessity of expanding the the AVAIL information to track call sites that occur between the definition of an expression and a later recomputation of it [Burk 83].

Two systems are closely related to this work. Feldman's *make* system is an ancestor of our system, since it pioneered the idea of automatic reconstruction based on an analysis of the internal dependences of a system [Feld 79]. However, *make* requires that the dependences be explicitly specified by the programmer, while our method automatically determines them. The system proposed by Tichy and Baker [TiBa 85] limits the recompilation required after a change to an *include* file by recording, for each module that uses it, which definitions are actually referenced. To determine which modules must be recompiled, it compares the stored information against the list of changed definitions. Although it is similar in flavor to our approach, the Tichy-Baker method cannot be used to limit recompilations that are forced by changes to interprocedural data flow information. Furthermore, the most advanced of our methods employs a more sophisticated approach to reasoning about what the compilation of a module actually uses.

The recompilation system reported here has been designed for implementation in the programming environment. A fundamental goal of the project is to experiment with interprocedural analysis and optimization. The environment assists in recompilation analysis in two ways: its database stores information for the compiler between compilations, and its source editor automatically records the information about individual procedures needed to make recompilation decisions.

Although the environment is designed to support Fortran, none of the techniques discussed in this paper are Fortran specific. They work with the data flow information produced by a compiler and are applicable across a wide variety of languages. The interprocedural techniques developed as part of all assume a language with recursion and all deal with block structured naming, making them applicable to most Algol-like languages.

2. Interprocedural Information

Familiarity with the interprocedural data flow information collected and used in the environment is a prerequisite to understanding the recompilation tests, so we begin with some background. Interprocedural information provides the

†This research has been supported by the National Science Foundation through grants MCS 81-21844 and MCS 83-03638 and by IBM Corporation.

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. The environment collects information on three distinct interprocedural phenomena: aliasing, side effects, and constant propagation.

Whenever two names can refer to a single storage location, they are potential *aliases*. Because an assignment actually modifies both the name and all of its aliases, the compiler needs reasonably precise information about potential aliases.[1] In the absence of such information, it must assume that all formal parameters and global variables are potentially aliased, in order to preserve the relative ordering of loads and stores. In practice, this eliminates opportunities for optimizations involving those variables. For example, the compiler cannot retain the values of global variables or formal parameters in registers. To provide the compiler with knowledge of aliasing patterns, the environment annotates each procedure p with a set ALIAS(p) containing those pairs of names that are aliased along some path leading to p [Coop 85].

Side effect summary information describes the side effects of a procedure call on the values of variables visible from the point of call. At a call site, executing the body of the called procedure can both reference and change the values of variables visible in the calling procedure. Since the compiler relies on derived knowledge about the values of variables to determine the safety and profitability of optimizations, it must understand the impact of a procedure call on the values of variables in the calling procedure. In the absence of precise information, it must assume that the call both modifies and uses every variable available to it. Because global analysis uses this information to summarize the effects of the call, this worst case assumption decreases the accuracy of the global data flow information computed for the calling procedure, inhibiting optimization. For example, in available expression analysis, a procedure call must be assumed to kill every expression involving either a global variable or an actual parameter. The programming environment provides the compiler with flow insensitive summary information by annotating each call site e in a program with two sets, MOD(e) and USE(e). The former contains all variables that can be modified as the result of executing e, while the latter contains all those variables whose values can be used as the result of executing e. It computes this information by solving a *backward* data flow analysis problem on the program's call graph [CoKe 84].

In large programs, information is often passed between procedures in the form of constant-valued actual parameters. This is

```
procedure a;
  begin;
      declare (x,y,z) fixed bin external;
        ...
      local2 = 17;
  α:  call c(x, local2);
        ...
  β:  call b(local1, local2);
        ...
  end;

procedure b(parm1, parm2);
  begin;
      declare (parm1, parm2) fixed bin;
        ...
  γ:  call c(parm1, parm2);
      parm2 = parm1 * 3;
        ...
  end;

procedure c(parm3, parm4);
  begin;
      declare (x,y,z) fixed bin external;
      declare (parm3, parm4) fixed bin;
        ...
      parm3 = parm4 * 2;
        ...
  end;
```

Figure 1. Example program fragment.

[1]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 [ANSI 78]. Nevertheless, the compiler attempts to trace aliases because information about potential aliases can be useful as a diagnostic aid to the programmer and because we want our system to achieve a higher level of predictability than the standard requires.

particularly common in numerical programs that incorporate modules from standard libraries like LINPACK [DBMS 79], 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 that always receive the same constant value. Finding a precise solution to the general constant propagation problem is undecidable [KaUl 77] and the usual approximate constant propagation problem is intractable in an interprocedural setting [Myer 81]. Therefore the environment relies on algorithms that efficiently compute approximations to the sets of constant parameters. The programming environment computes, for each procedure p in the program, a set CONSTANTS(p) of constants known to hold on entry to p. Elements of CONSTANTS(p) are tuples of the form <x,v>, where x is the name of a formal parameter or global variable and v is its known constant value. A companion paper describes a family of algorithms for solving this problem [CCKT 86, Torc 85].

As an example, consider the program fragment shown in Figure 1. Assuming that all of the relevant statements are shown, the aliasing and constants sets for its procedures would be:

procedure	ALIAS	CONSTANTS
a	∅	∅
b	∅	{ <*parm*2, 17> }
c	{ <*x*, *parm*3> }	{ <*parm*4, 17> }

The potential alias for procedure c arises when call site β passes the global variable x as an actual parameter. The constants come about from passing the constant valued variable *local*2 as an actual at α and β; γ simply passes the value through to procedure c. The summary sets for the program fragment would be:

call site	MOD	USE
α	{ *x* }	{ *local*2 }
β	{ *local*1, *local*2 }	{ *local*1, *local*2 }
γ	{ *parm*1 }	{ *parm*2 }

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

3. The General Framework

All of our techniques for recompilation analysis apply the same test to determine when a procedure must be recompiled. The test compares the actual interprocedural information against *annotation sets*. These sets contain those interprocedural facts that can be true without invalidating the procedure's previous compilation. The three methods differ in the precision with which they assign values to these annotation sets. We attach the following sets to the program's call graph:[2]

(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 graph call 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) *MayUse*(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 USE(e) that is not in *MayUse*(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 there exists a pair <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 any of the following are true:

(a) ALIAS(p) - *MayBeAlias*(p) ≠ ∅

(b) MOD(e) - *MayMod*(e) ≠ ∅, for any call site e in p

(c) USE(e) - *MayUse*(e) ≠ ∅, for any call site e in p

[2]The call graph is actually a multi-graph, with an edge for every call site. For the sake of consistency with the literature and readability, we refer to it as a graph.

(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 analyzer first initializes the list to include every procedure where editing has produced a semantic change since its last compilation. Next, it applies incremental techniques to update the program's ALIAS, MOD, USE, and CONSTANTS sets. 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

$$MayBeAlias(p) = \emptyset \text{ and}$$
$$MustBeConstant(p) = \{ \langle x,\Omega \rangle, \text{ for all } x \in p \},$$

where x ranges over the parameters and global variables of p and Ω is a constant value that appears nowhere in the program, and for each call site e let

$$MayMod(e) = \emptyset \text{ and}$$
$$MayUse(e) = \emptyset.$$

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 test which recompiled 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: MOD(β) = {*local*1} and USE(β) = {*local*2}. 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 depends entirely on the values assigned to *MayBeAlias*, *MayMod*, *MayUse*, and *MustBeConstant*. To improve the precision of the test involves expanding *MayBeAlias*, *MayMod*, and *MayUse* to include more allowed facts, or shrinking *MustBeConstant* to exclude more facts. The next three subsections present different methods of computing values for these sets. The methods are presented in increasing order of complexity; each successive method gives rise to a recompilation analysis of improved precision.

3.1. Most Recent Compilation

Our first approach to computing the annotation sets simply remembers the values of ALIAS, MOD, USE and CONSTANTS used in the most recent compilation of the procedure. In other words, whenever we compile a procedure p, we set

(1) *MayBeAlias*(p) = ALIAS(p),

(2) *MayMod*(e) = MOD(e), for each call site e in p,

(3) *MayUse*(e) = USE(e), for each call site e in p, and

(4) *MustBeConstant*(p) = CONSTANTS(p).

This set of assignments reveals the principles underlying the recompilation tests. The summary and aliasing sets are called *flow insensitive* because they identify events whose occurrence *cannot be ruled out* by the analysis. In other word, in considering the ALIAS, MOD, and USE sets, the compiler can only depend on what is *not* in the sets. 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. 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 may not occur. Hence, changes in flow insensitive information necessitate recompilation only when they involve *additions* to those sets. Although a deletion might create a new opportunity for optimization, it cannot invalidate the correctness of previous compilations. This principle motivates tests (a), (b), and (c).

On the other hand, the CONSTANTS(p) set approximates a flow sensitive set, so changes to it

must be handled differently. Flow sensitive sets are used to assert that a fact holds along every path leading to some point in the program. For example, 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 change in the program later causes <x,v> to be removed from CONSTANTS(p), such optimizations are invalidated. Thus, recompilations due to changes in flow sensitive sets are required only in response to *deletions*. Additions can provide new opportunities for optimization, but will not invalidate previously applied transformations. 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(β) and USE(β). 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.

3.2. Reference 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 advantage of the availability of the source text for the procedure under consideration. In particular, it recompiles a procedure whenever a variable is added to one of its flow insensitive annotation sets, even if that variable is not actually referenced anywhere in the procedure. Determining which variables are actually referenced leads immediately to an improved test. If the recompilation analysis is performed in a programming environment, the editor can easily produce such information.

To describe the annotation sets for this improved test, we define three additional sets. For a procedure p, REF(p) is the set of variables either used or modified inside p. REF$^+$(p) is defined to be the set of all variables referenced in p or some procedure invoked as a result of executing p. Both REF(p) and REF$^+$(p) can be computed trivially from information produced in the summary computation. Finally, the set AliasREF(p) describes pairs of variables, one of which is referenced locally in p and the other referenced in p or one of the procedures that can be executed as a result of invoking p. This set is defined as

$$\text{AliasREF}(p) = \{ < x, y > \mid x \in \text{REF}(p) \text{ and } y \in \text{REF}^+(p) \}$$

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

(1) *MayBeAlias*(p) = ALIAS(p) $\cup$ $\overline{\text{AliasREF}(p)}$,

(2) *MayMod*(e) = MOD(e) $\cup$ $\overline{\text{REF}(p)}$,

(3) *MayUse*(e) = USE(e) $\cup$ $\overline{\text{REF}(p)}$, and

(4) *MustBeConstant*(p) = CONSTANTS(p) $\cap$ REF(p). [3]

Computing the annotation sets from these definitions eliminates spurious recompilations that arise from information about irrelevant variables. In practice, this is important because procedures often contain declarations for global variables they never reference. In Fortran, this happens with COMMON statements; in other languages, widespread use of include files achieves the same result.

To see this more graphically, consider adding the statement

x = *parm*4 * 17;

to procedure c in the ongoing example. This changes MOD(γ) to {*parm*1, x} and MOD(β) to {*local*1, x}. Under the most recent compilation test, this would have required recompilation of both a and b. Using reference information, the test determines that a requires recompilation, but b does not, since b doesn't reference x.

The equations are presented in this form to convey the underlying ideas; an actual implementation should avoid instantiating sets like $\overline{\text{REF}(p)}$ and $\overline{\text{AliasREF}(p)}$. A careful refactoring of the equations leads directly to a much more efficient implementation.

---

[3] The intersection in definition (4) is not intended to be taken literally; CONSTANTS(p) contains <name,value> pairs while REF(p) contains only names. Our intent is to compute the set of pairs in CONSTANTS(p) where the name is also a member of REF(p). This operation is a *join* over REF(p) in relational algebra.

3.3. Compiler Cooperation

While the use of reference information eliminates many spurious recompilations, some can still occur because the compiler cannot use every interprocedural fact. Thus, a fact judged to mandate recompilation by the reference test may actually be irrelevant, simply because the compiler was unable to capitalize on it to justify an optimization during the most recent compilation. This section explores a technique for computing exact annotation sets in which the compiler records which interprocedural facts it depends upon so that the recompilation analyzer can more precisely determine whether a recompilation is actually needed.

We begin by considering two sample optimizations and the global data flow information required to support them, in an attempt to discern how to compute the contribution to annotation sets for these optimizations.[4] Throughout this discussion, a procedure p will be represented by its data flow graph, $G = (N,E,n_0)$. The nodes of this graph represent *basic blocks*, sequences of statements with no control flow branches. The edges $e = (m,n) \in E$ represent *control flow* between two basic blocks. Control enters the procedure through its entry node n_0.

3.3.1. 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. 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:

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

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 [KaUl 76], 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, 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 MOD(e) for the call site e. USE(e) plays no role in the AVAIL computation.

When a variable $v \in$ MOD(e), no expression containing v can be in 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 $\alpha \in$ AVAIL(b) for some block b, its constituent variables cannot be in the MOD set of any call site between α's most recent evaluation and b, on each path leading to b. If the compiler eliminates a re-evaluation of α, 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 α are added to one of these MOD sets.

To capture this information in the annotation sets, the compiler must determine the set of call sites, CALLS_BETWEEN(α,b), between the last computation of an expression α and the block b where the re-evaluation is replaced, along each path leading to b. Given CALLS_BETWEEN(α,b), *MayMod*(e) can be constructed as follows:

[4] The equations are taken from Kennedy's survey paper [Kenn 81].

(1) *MayMod*(*e*) = 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 α is replaced in block *b*, the compiler removes all constituent variables of α, from *MayMod*(*e*), for each call site *e* in CALLS_BETWEEN(α,*b*) and each call site *e* inside *b* occurring before the optimization.[5]

The resulting *MayMod* sets precisely describe the recompilation dependences introduced by applying this optimization.

Now we turn to the question of how to compute CALLS_BETWEEN(α,*b*). This requires solving an auxiliary data flow problem. Replace the elements of AVAIL(*b*), DEF(*b*), and NKILL(*b*) with tuples of the form <*name,calls*> where *name* is the literal name associated with the available expression and *calls* is defined as follows:

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

The necessary *calls* sets can be accumulated during the AVAIL computation by carefully redefining the operators and local sets as follows.

X∩Y: To compute X∩Y, for each element *x* from X such that there exists a $y \in$ Y with $x.name = y.name$, add $<x.name, x.calls \cup y.calls>$ to the result.

X∪Y: To compute X∪Y, partition all the elements of X and Y into four sets: *Xonly*, containing every element whose name appears only in X, *Yonly*, containing every element whose name appears only in Y, *Xboth*, containing every element of X for which there is an element of Y with the same name, and *Yboth*, containing every element of Y for which there is an element of X with the same name. Let *XY* be computed by taking *Xboth*∩*Yboth* using the procedure above. Then the desired result is the natural union of *Xonly*, *Yonly* and *XY*.

Using these definitions, for each $\alpha \in$ AVAIL(*b*), α*.calls* corresponds to the set CALLS_BETWEEN(α,*b*). Even with the changes in the operators and local sets in the AVAIL computation, calculation of the new AVAIL and CALLS_BETWEEN(α,*b*) information is still *rapid* in the sense of Kam and Ullman [KaUl 76].

3.3.2. 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 on entry to *b*. LIVE sets can be computed by solving the following backward data flow problem:

$$\text{LIVE}(b) = \text{IN}(b) \cup \bigcup_{a \in S(b)} (\text{THRU}(b) \cap \text{LIVE}(a))$$

In this equation, *S*(*b*) is the successor set of *b*. IN(*b*) is the set of variables used in *b* before being redefined. Variables in IN(*b*) are live on entry to *b*. THRU(*b*) is the set of variables not redefined in *b*.

Without summary information about procedure calls, the compiler must assume that a procedure call uses any variables visible to it. This assumption can extend the live ranges of variables, inhibiting the application of register store elimination. Interprocedural USE sets can reduce the set of variables assumed LIVE because of a call site. Because MOD(*e*) says nothing about the ordering of uses and definitions, 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

[5] The optimizer has assumed these variables are not in MOD(*e*) at each of these call sites.

end of the procedure can potentially invalidate a register store optimization. Adding a variable to the USE set of such a call site would make the eliminated store necessary for correct execution of the program. Assume the existence of a set CALLS_AFTER(b) for each block b, containing the set of call sites in the procedure containing b that can be executed after execution of b.

To construct a recompilation test that precisely characterizes the use of interprocedural information in the register store optimization, we must compute a larger *MayUse*(e) set. Given this set, *MayUse*(e) can be computed as follows:

(1) *MayUse*(e) = ALLVARS, for all call sites e;

(2) Whenever a store of a variable v is eliminated, the optimizer removes v from *MayUse*(e) for each call site e in CALLS_AFTER(b) and each call site inside b occurring after the optimization.

This results in *MayUse* sets that precisely capture the recompilation dependences for this optimization.

There is an important special case of step (2) above for languages like Fortran or C that do not have nested name scopes. The optimizer need only remove a local variable v (e.g., not in COMMON) from *MayUse*(e) if e passes v as an actual parameter. This observation should decrease the number of set operations required to construct precise *MayUse* sets.

To compute CALLS_AFTER(b), the following system of data flow equations must be solved:

$$\text{CALLS\_AFTER}(b) = \bigcup_{a \in S(b)} (\text{L\_CALLS}(a) \cup \text{CALLS\_AFTER}(a))$$

where L_CALLS(a) is the set of call sites in basic block a. CALLS_AFTER(b) is the set of call sites that occur after basic block b. This calculation is *rapid* in the sense of Kam and Ullman [KaUl 76].

3.3.3. Generalization

Examination of the two 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×ALLVARS

(2) *MayMod*(e) = ALLVARS, for each call site e in p

(3) *MayUse*(e) = ALLVARS, for each call site e in p

(4) *MustBeConstant*(p) = ∅

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

The discussion of the two optimizations should demonstrate the difficulties of determining, for each call site, which assumptions can invalidate an optimization. It depends upon the particular optimization and usually requires the solution of an auxiliary data flow problem. Computing precise annotation sets will likely increase the compile times for individual modules. We hope that effective interprocedural optimization will mitigate the increased compile time, by making individual procedures smaller and localizing recompilation more precisely.

This section showed an approach for computing exact recompilation information for changes in flow insensitive summary sets. Determining which procedures need recompilation due to changes in CONSTANTS sets is easier. Understanding how the compiler actually uses CONSTANTS information is crucial. For a procedure p, 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 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* whenever it folds v into a use of x.

The interprocedural constant analysis can also produce sets describing constant values returned by procedures through global variables, and call-by-reference formal parameters [CCKT 86]. 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 that

required for AVAIL.

For aliasing information, it appears that there is no reason to construct a precise test. This is true, in large part, because of the manner in which aliasing information is used. 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. Thus, for aliasing, the test based upon references is as good as we can do in a reasonably efficient compiler.

4. Larger Compilation Units

The treatment presented in Section 3 implicitly assumes that each compilation unit contains only one procedure. Many compilers allow multiple procedures to be treated as an indivisible compilation unit; in , we call such a unit a *module*. The presence of multiple procedures in a single module slightly complicates the recompilation analysis. When analyzing a module that contains multiple procedures, the compiler must recognize that the procedures are related. Once it marks a single procedure for recompilation, it can ignore all the other procedures in the module, since they will be recompiled, too.

To maintain these relationships, the compiler must build a simple look-aside table that maps a procedure into a module and a module into a set of procedures. Using the table, the recompilation analyzer then marks all of the procedures in a module for recompilation whenever any of its constituent procedures needs recompilation. This has the potential to decrease the total amount of work required for the analysis. As a module is marked for recompilation, it eliminates multiple procedures from consideration, possibly decreasing the total number of procedures analyzed.

It is important to recognize the difference between this approach and a hierarchical approach like that found in structural data flow algorithms. This approach maintains separate data flow information for each of the procedures, but accounts for the textual relationships between them. Merging the nodes for the procedure would simplify the graph, but would result in merging the information used in the test and losing the path sensitivity of the 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.

This technique for handling multiple procedures in a single module leads to a natural mechanism for handling the impact of interprocedural optimizations on the recompilation analysis. Optimizations like linkage tailoring or common subexpression elimination across procedure boundaries clearly introduce new compilation dependences between the involved procedures. The discussion of Section 3 completely ignores them. Whenever the compiler applies an interprocedural optimization, it can simply mark the involved procedures to be treated as members of a single module. This requires adding entries to the two maps just described. Thus, a change to one procedure results in recompiling both and reconsidering the optimization.

5. Improved Optimization

We have seen that changes in interprocedural information can invalidate the safety of optimizations applied in previous compilation. For flow sensitive summary and aliasing sets, adding facts to a set associated with a procedure mandated recompiling it, while deleting facts did not. Deletions do, however, open up new possibilities for applying optimizations. Recall that optimizations based on MOD, USE, 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.

The recompilation tests presented in Section 3 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. Unfortunately, the decision to recompile to obtain improved optimization is not as straightforward as the decision for correctness. The improved interprocedural environment may not lead to better optimization for any number of reasons. Further, the expense of recompiling must be weighed against any improvement. A method of

estimating improvement based on specific interprocedural facts is needed, similar to that proposed by Ball for constant information [Ball 79].

6. 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 module compilation time versus number of spurious recompilations.

These methods are being implemented as a part of the programming environment project at Rice University.

7. Acknowledgements

The original work on this problem was inspired by a question posed by Michael Burke of IBM Research, who actively participated in early discussions of the problem. Frances Allen of IBM also contributed to these discussions. The implementation team has provided a marvelous research vehicle for experimenting with new ideas about interprocedural analysis and optimization. To all of these people go our heartfelt thanks.

8. References

[AlCa 80] F.E. Allen *et al*. The experimental compiling system. *IBM Journal of Research and Development*, 24(6), 1980.

[ANSI 78] American National Standards Institute. *American National Standard Programming Language Fortran, X3.9-1978*.

[Ball 79] J.E. Ball. Predicting the effects of optimization on a procedure body. *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction*, SIGPLAN Notices, 14(8). 1979.

[Burk 83] M. Burke. Private Communication. November, 1983.

[CCKT 86] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. SIGPLAN Notices, 21. 1986.

[Coop 85] K.D. Cooper. Analyzing aliases of reference formal parameters. *Proceedings of the Twelfth POPL*. January 1985.

[CoKe 84] K.D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*. SIGPLAN Notices, 19(6). 1984.

[CoKT 86] K.D. Cooper, K. Kennedy and L. Torczon. Optimization of compiled code in the programming environment. *Proceedings of the Nineteenth Annual Hawaii International Conference on Systems Sciences*, 1986.

[Conr 83] R. Conradi. Inter-procedural optimization of object code. TR 25/83, Division of Computer Science, University of Trondheim, Trondheim-NTH, Norway. 1983.

[DBMS 79] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia. 1979.

[Feld 79] S. Feldman. Make - a computer program for maintaining computer programs. *Software Practice and Experience 9*, 1979.

[KaUl 76] J. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1). January 1976.

[KaUl 77] J. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7. 1977.

[Kenn 81] K. Kennedy. A survey of data flow analysis techniques. In *Program Flow Analysis: Theory and Applications*, New Jersey: Prentice-Hall. 1981.

[Myer 81] E.W. Myers. A precise inter-procedural data flow algorithm. *Proceedings of the Eighth POPL*. January 1981.

[Spil 71] T.C. Spillman. Exposing side-effects in a PL/I optimizing compiler. *IFIPS Proceedings*, 1971.

[TiBa 85] Tichy W. F. and M. C. Baker, ‘‘Smart Recompilation’’, *Proceedings of the Twelfth POPL*, 1985.

[Torc 85] L. Torczon. Compilation dependences in an ambitious optimizing compiler. Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX. May 1985.