Interprocedural Optimization: Eliminating Unnecessary Recompilation

Michael Burke
Keith D. Cooper
Ken Kennedy
Linda Torczon

CRPC-TR90058
July, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Interprocedural Optimization: Eliminating Unnecessary Recompilation†

Michael Burke
IBM T. J. Watson Research

Keith D. Cooper, Ken Kennedy, 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: compilers, languages, performance

Additional Keywords and Phrases: data-flow analysis, recompilation analysis, interprocedural analysis and optimization

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 interprocedural analysis and optimization. Recent work has included both faster algorithms for some interprocedural data-flow problems and efficacy studies of some interprocedural transformations [BuCy 86, CCKT 86, Call 88, Chow 88, CoKe 88, Wall 88, CoKe 89, RiGa 89a, RiGa 89b]. 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 FORTRAN and 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

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

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 dependences, 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 recompiled unnecessarily.

This paper examines the recompilation problems introduced by the use of interprocedural data-flow information as a basis for compile-time decisions. 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 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 for each executable [AlCa 80]. Some systems ignore the problem completely — for example, the Cray FORTRAN compiler will perform inline substitution, but doesn't provide the user with support for managing the resulting recompilation problems.

However, there are two systems that are closely 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 [Feld 79]. However, *make* requires that the programmer explicitly specify the compilation dependences, while our method derives them analytically. The system proposed by Tichy and Baker [TiBa 85] 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. 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.

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 distinct 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 several implementation techniques. Section 5 proposes a compiler implementation methodology 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 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

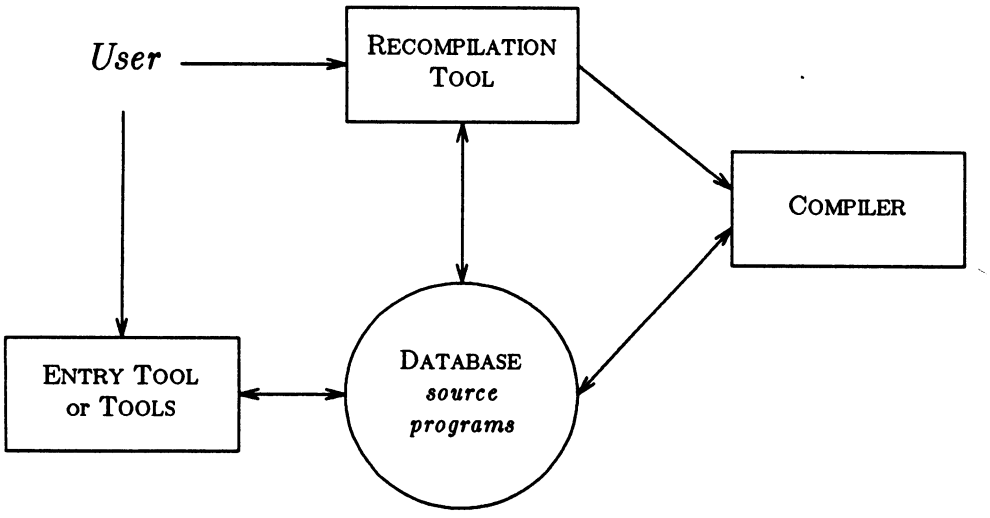

Figure 1 — Our compilation model

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 $\mathbb{R}^n$ programming environment for FORTRAN is an example [CCHK 87]. Similarly, an implementation within a more conventionally structured compiler can fit the model — the PTRAN analysis system for FORTRAN is an example [ABCC 88]. 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. Thus, we are concerned with 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 obtain knowledge of aliasing patterns, the compiler can compute, for each procedure p, 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 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, it must understand the impact of a procedure call on the values of variables in the calling procedure. 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.

---

[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, 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, MOD(e) and REF(e).[2] 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 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 [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 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 [KaUl 77] and the usual approximate constant propagation problem is intractable in an interprocedural setting [Myer 81]. 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 [Torc 85, BuCy 86, CCKT 86, WeZa 88]. In our model system, the compiler 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 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.

```
      program a                    subroutine b(p1, p2)    subroutine c(p3, p4)
      integer x,y,z,v1,v2          integer p1, p2          integer x,y,z,p3,p4
      common /global/x,y,z           ...                   common /global/x,y,z
        ...                  γ:    call c(p1, p2)            ...
      v2 = 17                      p2 = p1 * 3             p3 = p4 * 2
α:    call c(x, v2)                  ...                     ...
        ...                        end                     end
β:    call b(v1, v2)
        ...
      v2 = v2 * x
        ...
      end
```

Figure 2. Example program fragment.

[2] For consistency with the rest of the literature on interprocedural data-flow analysis, we will call this set REF, even though we have used the name USE in the past. USE appears in several sources as the set of variables whose values can be read before modification. 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 REF set is inherently flow-insensitive, while the USE set is inherently flow-sensitive.

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:

procedure	ALIAS	CONSTANTS
a	$\emptyset$	$\emptyset$
b	$\emptyset$	{ **(p2,17)** }
c	{ **(x,p3)** }	{ **(p4,17)** }

The potential alias for procedure **c** arises when call site α passes the global variable **x** as an actual parameter. The constants come about from passing the constant valued variable **v2** 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	REF
α	{ **x** }	{ **v2** }
β	{ **v1,v2** }	{ **v1,v2** }
γ	{ **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 is recompiled if 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

(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 editing has produced a semantic change since its last compilation. Next, it applies incremental techniques to update the program's ALIAS, MOD, REF, 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) $=\emptyset$ and

MustBeConstant(p) $=\{ (x,\Omega)$, for all x declared in the program$\}$,

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) $=\emptyset$ and

MayRef(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 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: MOD(β) $=\{$**v1**$\}$ and REF(β) $=\{$**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. The next three sections 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.

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 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) *MayRef*(e) =REF(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 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 may 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(β) and REF(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**.

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 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$^{\uparrow}$(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^{-1}(p) can be computed trivially from information produced in the summary computation. Finally, the set ALIASAPPEARS(p) describes pairs of variables, one of which appears locally in p and the other appears in p or one of the procedures that can be executed as a result of invoking p. This set is defined as

$$\text{ALIASAPPEARS}(p) = \{(x,y) \mid x \in \text{APPEARS}(p) \text{ and } y \in \text{APPEARS}^{-1}(p)\}$$

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

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

(2) *MayMod*(e) = MOD(e) $\cup$ $\overline{\text{APPEARS}(p)}$, for each e in p,

(3) *MayRef*(e) = REF(e) $\cup$ $\overline{\text{APPEARS}(p)}$, for each e in p, and

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

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 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 achieves the same result. 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 doesn't involve declarations that are actually relevant to the procedure [TiBa 85].

To see this more graphically, consider adding the statement

```
x = p4 * 17
```

to procedure c in the example from Figure 2. This changes MOD(γ) to {**p1**,**x**} and MOD(β) 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** doesn't appear in **b**.

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

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.

This distinction gives rise to the issue of "hidden variables" described by Burke and Cytron [BuCy 86]. As we have formulated them, the recompilation tests determine when an intraprocedural optimization can be invalidated by a subsequent change in an interprocedural set. The APPEARS test will not handle interprocedural optimizations correctly, when it is applied in a straightforward manner. Such optimizations require a more complex treatment; we describe one such method in

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

Section 7. Note, also, that the simpler test described in Section 4.1 will produce correct results for such "hidden variable" problems.

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, 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. 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, Sethi, and Ullman [AhSU 86]. Consider equations of the form:

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

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, $\bigcup$ or $\bigcap$. 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.

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 event α and a block b in the control-flow graph where the presence of α is used to justify an optimization. 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 event α and block b, along all paths leading from α to b. We will call that set CALLSBETWEEN(α,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 event α, along all paths leading from b to α. We will call that set CALLSBETWEEN(b,α). For the sake of simplicity, we will refer to both types of information as 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 $gen(a)$ is defined, whether $f(b)$ is $P(b)$ or $S(b)$, and the region of interest considered for CALLSBETWEEN.

To compute 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 events. Thus, the presence of an element α in $out[b]$ asserts some property that holds true at b. In the auxiliary data-flow problem, the presence of α in $out[b]$ still denotes that α is a data-flow event holding at b, but it is now represented in terms of a pair ($\alpha.name$,$\alpha.calls$), where $\alpha.name$ represents the literal name of the event and $\alpha.calls$ is CALLSBETWEEN. To solve this auxiliary problem, we expand the domain of the *gen* and *nkill* sets accordingly:

- For $\alpha \in gen[b]$, if the data-flow problem is a *forward* problem, $\alpha.calls$ is the set of call sites in b *after the last* occurrence of α; if it is a *backward* problem, $\alpha.calls$ is the set of call sites in b *before the first* occurrence of α.
- For $\alpha \in nkill[b]$, $\alpha.calls$ 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 $x.name = y.name$, add ($x.name$,$x.calls \cup y.calls$) to the result.

$X \cup Y$ 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*.[4]

$X \bigwedge Y$ To compute $X \bigwedge Y$, if the appropriate meet operation is $\bigcup$, 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 $\bigcap$, 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 data-flow 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

[4] Note that $X \cup Y$ as defined here is not a commutative operation. For this to work correctly, X must represent $gen[a]$ and Y must represent $(out[a] \cap nkill[a])$.

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 [AhSU 86]. 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:

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

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. REF(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 can compute CALLSBETWEEN sets along with the AVAIL information and use them to compute *MayMod*(e). The CALLSBETWEEN sets are computed as described in Section 5.1. For each $\alpha \in$ AVAIL(b), α.*calls* is CALLSBETWEEN(α,b). The following definitions are used for the local sets.

- 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.

The operations used are those described in Section 5.1. In this case, the meet operator is intersection. Using these definitions, for each $\alpha \in$ AVAIL(b), α.*calls* corresponds to the set CALLSBETWEEN(α,b). Even with the changes in the operators and local sets in the AVAIL computation, calculation of the new AVAIL and CALLSBETWEEN information is still *rapid* in the sense of Kam and Ullman [KaUl 76].

Given CALLSBETWEEN(α,b), *MayMod*(e) can be constructed as follows:

(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 CALLSBETWEEN(α,b) and each call site e inside b occurring before the optimization.[5]

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

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 [AlCo 72]. 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 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 set of *very busy expressions*.

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

Here, USED(a) contains those expressions computed in a prior to redefinition in a of any of its constituent variables.

When a variable $v \in$ MOD(e), no expressions containing v can be in NKILL(b) for the block b containing e. Thus, if an expression $\alpha \in$ VERYBUSY(b) for some block b, its constituent variables cannot be in the MOD set of any call site between the end of b and the first evaluation of α on each path leading from b. To apply the hoisting optimization, the compiler would move the evaluation of α 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 α relies on the values of the 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 α are added to one of these MOD sets.

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

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

The meet operator is intersection. The data-flow problem is still *rapid* in the sense of Kam and Ullman, even after the addition of the auxiliary problem [KaUl 76].

---

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

Given CALLSBETWEEN(b,α), *MayMod*(e) can be updated for code hoisting in the following manner.

(1) *MayMod*(e) =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 α to the end of block b, the compiler should remove each of the constituent variables of α from *MayMod*(e) for each α in CALLSBETWEEN(b,α).

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.[6] 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.

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

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. The meet operator is set union.

However 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.

Note, however, that while a new definition in one of these MOD sets invalidates the actual folding of c at p, it doesn't actually invalidate the REACH sets. The safety of the folding transformation is based on a stronger condition than the presence a definition in REACH(b). The constant can be folded only if, along all paths, all definitions of x that reach b have the known constant value c. Thus, adding x to MOD(e) for some call site e produces a new definition of x that can invalidate the condition for any block b that the new definition reaches.[7]

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(α,b). In this case, we use the following

[6] Where interprocedural constant propagation is performed, the CONSTANTS sets are used as initial information in the single procedure, or global, computation. In Section 5.7, we consider the precise computation of *MustBeConstant* with respect to global constant propagation.

[7] Recall that a definition in MOD represents a data-flow event that occurs along some execution path from the procedure call, but not necessarily along *all* paths from the call. Thus, a definition of v must be treated as preserved (*not* killed) by a call site e, even if $v \in$MOD(e), since there can be a path through the called procedure that does not include the modification of v. Thus, adding v to MOD(e) doesn't invalidate the REACH sets; the reaching characteristics of the other definitions in the program are unchanged.

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 α.
- For $\alpha \in$ NKILL(b), $\alpha.calls$ is the set of all call sites in b.

With these definitions, the revised REACH equations will compute both reaches information and the CALLSBETWEEN sets.[8] The revised computation is still rapid in the sense of Kam and Ullman [KaUl 76].

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

(1) *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(α,b) for each definition α 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.[9]

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 don't 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 [CCKT 86].

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 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,α) now describes a region between b and a point at which the optimizer decided that some event involving α *did not* occur. In the case of register store elimination, if α is not LIVE at b, CALLSBETWEEN contains all call sites between block b and a redefinition of α (or an exit from the procedure if no redefinition exists) along every path leaving b.

These differences are troublesome; we would like to fit type II optimizations into the same basic framework as the type I optimizations.

The solution to this quandary is to coerce the type II optimizations into the form of type I optimizations. To do this, we simply compute the information that the optimizer really uses: $\overline{out[b]}$. When the optimizer relies on the absence of a fact from some data-flow set, we recast the problem to

[8] 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 define $X \cup Y$ as a natural union instead of the natural union of X and Y*only*. However, it is correct as defined here and it fits the proposed framework.

[9] 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.

compute the complement of that set, so that the transformation can be based on the presence of a fact in the complement.

To make this more concrete, we can recast our general formulation, equation (1) from Section 5, as follows.

$$out_0[b] = \bigwedge_{a=f(b)}{}_0 \, (gen_0[a] \cup (out_0[a] \cap nkill_0[a]))$$

Using DeMorgan's law, we compute the equation for $\overline{out_0[b]}$. Note that $\bigwedge$ is the dual of $\bigwedge_0$, where $\bigcup$ and $\bigcap$ are duals.

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

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

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

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

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

with the following assignments $out[b] = \overline{out_0[b]}$, $gen[a] = \overline{gen_0[a]} \cap \overline{nkill_0[a]}$, 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, Sethi, and Ullman [AhSU 86].

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

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 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 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 $out[b]=\overline{\text{LIVE}(b)}$, $gen[a]=\overline{\text{IN}(a)} \cap \overline{\text{NDEF}(a)}$, and $nkill[a]=\overline{\text{IN}(a)}$. Let the meet operation be set intersection. Now the general equation we gave in Section 5.0 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 $\alpha \in out[b]$, $\alpha.calls$ represents the set CALLSBETWEEN(b,α). The following definitions work within the general framework described in Section 5.1.

- For $\alpha \in gen[b]$, $\alpha.calls$ is the set of call sites in b *before the first* definition of α.
- For $\alpha \in nkill[b]$, $\alpha.calls$ is the set of all call sites in b.

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 [KaUl 76].

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) $MayRef(e)$ =ALLVARS, for all call sites e;

(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,α) and each call site inside b occurring after the optimization.

This results in *MayRef* sets that precisely capture the recompilation dependences 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 the beginning of the block; for a forward flow computation, it is the end of the block. When a fact α passes through a block unchanged, all of the call sites in the block are added to $\alpha.calls$ because the block is an α-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 $X \cup Y$ operator, which is interpreted as $X \cup Yonly$. 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 $X \cup Yonly$ 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 $X \cup Yonly$ operation because these are the events that happened nearest to the point of optimization. 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 2 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 some 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 $\overline{\text{LIVE}}$ 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.

	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	$\overline{any-path}$	backward

Figure 2 — Summary of examples

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, so, on the meet-over-all-paths solution for some data-flow problem) [Rose 80, Tarj 81].

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.[10]

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 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 $O(p\,E\,d(G))$ for reducible graphs and $O(p\,E\,N)$ for non-reducible 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 [KaUl 76].

If we assume that nested procedures always 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

---

[10] 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 doesn't change the values of any variable on return from the procedure. If we assign a variable its known constant value, we really don't 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.

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×ALLVARS

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

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

(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*.

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 of *MayMod* sets for common subexpression elimination and code hoisting illustrates the accommodation of this kind of impact.

This section showed an approach for computing more precise recompilation information for changes in MOD and REF 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 and VERYBUSY.

For aliasing information, it appears that there is no reason to construct a test that is more precise than the APPEARS test discussed in Section 4.2. 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 ALIASAPPEARS sets appears to be as good as we can do in a reasonably efficient compiler.

Computing annotation sets that actually reflect compile-time decisions 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.

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 are preserved across the call. 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.[11] 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, 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 (MOD(e)$\cup$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 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 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 *MayMod*(e) and *MayRef*(e). Similarly, for each refresh eliminated after e, it would need to remove the variable from *MayMod*(e).

---

[11] The optimizing compiler for $\mathbb{R}^n$ tries to keep all scalar values in registers. Non-aliased 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.

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 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 4.0. 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(p) $\neq \emptyset$

(b) *WereMod*(e) - MOD(e) $\neq \emptyset$, for any call site e in p

(c) *WereRef*(e) - REF(e) $\neq \emptyset$, for any call site e in p

(d) CONSTANTS(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 shows a method for computing these annotation sets.

8.1. Computing the Annotation Sets

In Section 4.2, we showed a method for computing approximate annotation sets for the correctness test based purely on static information. Approximate annotation sets for the improvement test can be computed in a similar manner. At each compilation of a procedure p, the compiler can 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 compiler can compute:

(1) *WereAliased*(p) = ALIAS(p)$\cap$ALIASAPPEARS(p),

(2) *WereMod*(e) = MOD(e)$\cap$APPEARS(p), for each call site e in p,

(3) *WereRef*(e) = REF(e)$\cap$APPEARS(p), for each call site e in p, and

(4) *WereConstant*(p) = CONSTANTS(p)$\cup\overline{\text{APPEARS}(p)}$.

The rationale for these assignments is analogous to that underlying the correctness test in Section 4.2.

It doesn't 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; after all, incorrect code has no real value. 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 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.

10. Acknowledgements

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. Both the $\mathbb{R}^n$ and PTRAN implementation teams have provided marvelous research vehicles for for experimenting with new ideas about interprocedural analysis and optimization. To all of these people go our heartfelt thanks.

11. References

[AhSU 86] A.V. Aho, R. Sethi, J.D. Ullman. Compilers: principles, techniques and tools. Addison Wesley, 1986.

[ABCC 88] F.E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. "An overview of the PTRAN analysis system for multiprocessing", *Journal of Parallel and Distributed Computing* 5(5), October, 1988, pp. 617-640.

[AlCa 80] F.E. Allen, J.L. Carter, J. Fabri, J. Ferrante, W.H. Harrison, P.G. Loewner, L.H. Trevillyan. The experimental compiling system. *IBM Journal of Research and Development*, 24(6), 1980.

[AlCo 72] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and optimization of compilers*, J. Rustin(ed). Prentice-Hall, Englewood Cliffs, NJ, 1972.

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

[BuCy 86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, SIGPLAN Notices, 21(7), July, 1986.

[Call 88] D. Callahan. The program summary graph and flow sensitive interprocedural data flow analysis. *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7), July, 1988.

[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(7), July, 1986.

[CCHK 87] A. Carle, K.D. Cooper, R.T. Hood, K. Kennedy, L. Torczon, and S.K. Warren. A practical environment for scientific programming. *IEEE Computer* 20(11), November, 1987.

[Chow 88] F.C. Chow. Minimizing register usage penalty at procedure calls. *Proceedings SIGPLAN 88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7), July, 1988.

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

[CoKe 88] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation.* SIGPLAN Notices, 23(7), July, 1988.

[CoKe 89] K.D. Cooper and K. Kennedy. Fast interprocedural alias analysis. *Proceedings of the Sixteenth POPL*, January, 1989.

[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.

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

[RiGa 89a] S. Richardson and M. Ganapathi. Code optimization across procedures. *IEEE Computer* 22(2), February, 1989.

[RiGa 89b] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *To appear, Information Processing Letters.*

[Rose 80] Rosen, B.K. "Monoids for rapid data flow analysis". *Siam Journal of Computing* 9(1), February, 1980, pp. 159-196.

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

[Tarj 81] Tarjan, R.E. "A unified approach to path problems". *Journal of the ACM* 28(3), July, 1981, pp. 557-593.

[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.

[Wall 88] D.W. Wall. Register windows versus register allocation. *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7), July, 1988.

[WeZa 88] M. Wegman and F.K. Zadeck. Constant propagation with conditional branches. Computer Science Technical Report CS-88-02, Brown University, February, 1988.

Interprocedural Optimization: Eliminating Unnecessary Recompilation

Michael Burke
Keith D. Cooper
Ken Kennedy
Linda Torczon

CRPC-TR90058
July, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892