FIAT: A Framework for Interprocedural Analysis and Transformation

Alan Carle Mary Hall
John Mellor-Crummey Rene Rodriguez

CRPC-TR95522-S
March, 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

FIAT: A Framework for Interprocedural Analysis and Transformation

Mary W. Hall* John M. Mellor-Crummey† Alan Carle† René G. Rodríguez†

Abstract

Modern architectures with deep memory hierarchies or parallelism require the use of increasingly sophisticated code analysis and optimization to achieve maximum performance for large, scientific programs. In such programs, procedure boundaries can obscure an analysis system's understanding of program behavior and interfere with optimization. To address this need, we are building FIAT — a framework for interprocedural analysis and transformation that enables rapid prototyping of interprocedural techniques. Components of the framework include an interprocedural data-flow analysis engine, a procedure cloning analysis engine and parameterized templates for data-flow problems.

FIAT is being used to drive interprocedural optimization in the ParaScope programming tools at Rice University and the SUIF compiler at Stanford University. FIAT's use of system-independent abstractions makes it suitable for use in systems with distinct intermediate representations of programs and enables sharing of code across research platforms. Demand-driven analysis maintains a clean separation between interprocedural problems and enables tools built upon FIAT to solve only the problems of immediate interest. FIAT has proven to be an effective aid in development of a large number of interprocedural tools, including completed systems for data race detection and static performance estimation, and partial implementations of a distributed-memory compiler for Fortran D, an interactive parallelizing tool and applications within the SUIF compiler.

1 Introduction

To effectively exploit modern multiprocessor architectures, compilers and programming tools require deep program analysis to guide program transformations that expose coarse-grain parallelism and manage the memory hierarchy. A difficulty for such tools is that procedure boundaries can obscure their understanding of a program and interfere with transformations on loop nests spanning multiple procedures. Since large scientific codes typically contain many procedures, compilers and tools must employ *interprocedural* techniques to adequately support such programs. *Interprocedural data-flow analysis* crosses procedure boundaries to expose information about a procedure's calling context and the side-effects of its procedure calls. *Interprocedural transformations* restructure a program to reveal more precise data-flow information or move code across procedure boundaries.

Many researchers have demonstrated significant performance improvements on shared-memory multiprocessors by manually applying interprocedural analysis and transformation techniques to enhance parallelism or memory hierarchy utilization [5, 29, 49]. Techniques that have proven useful for this purpose include scalar and array side-effect analysis [31, 32, 39, 51], interprocedural constant propagation [23, 44], array KILL analysis [5, 26, 49], and transformation to expose loop nests to parallelization [29]. In addition to automatic parallelization, many problem domains in high-performance computing can greatly benefit from exploiting interprocedural information. For example, compiling for distributed-memory architectures and automatic detection of data races in shared-memory codes each involve program transformations that depend upon the

*Center for Integrated Systems, Stanford University, Stanford, CA 94305. This work was supported in part by ARPA contracts N00039-91-C-0138 and DABT63-91-K-0003.

†Center for Research on Parallel Computation, Rice University, Houston, TX 77251-1892. This work was supported in part by ARPA contract DABT63-92-C-0038. and NSF Cooperative Agreement Number CCR-9120008. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

solutions of domain-specific interprocedural problems for the generation of efficient code. Although interprocedural analysis and transformation have been shown to be useful, few compilers employ these techniques due to two major obstacles:

- Compilation and analysis must be fundamentally restructured to support interprocedural analysis and transformation. Interprocedural optimization of a procedure requires information gleaned from other procedures; thus, procedures cannot be compiled and optimized in isolation. Moreover, considerations of space and time complexity make straightforward extension of intraprocedural approaches infeasible; examining the entire program at once may consume too much space, and polynomial-time optimization algorithms may be prohibitively slow for large programs.
- A large programming effort is required to make effective use of interprocedural optimization. For example, determining the safety and profitability of interprocedural loop optimizations typically requires results from a wide range of interprocedural analyses. Without some common framework, solving each interprocedural data-flow problem would require a substantial amount of duplicated effort.

To address these obstacles, we are building FIAT, a framework for interprocedural analysis and transformation. FIAT provides an effective architecture for program analysis and restructuring in an interprocedural setting and serves as a vehicle for experimenting with interprocedural optimization and its applications. Moreover, FIAT's general framework greatly reduces the programming effort needed to incorporate new interprocedural data-flow analyses and transformations. FIAT currently supports interprocedural optimization in two very different host systems — the ParaScope programming tools at Rice University [16] and the Stanford SUIF Compiler [50]. This paper describes the following key aspects of FIAT.

Abstract representations. The representation of programs used within FIAT is decoupled from the representation used by any host system in which FIAT will be applied. This decoupling made it possible to extract FIAT from Rice's ParaScope system and retarget it to Stanford's SUIF compiler with only a small amount of recoding, resulting in sharing of analysis code across research projects. Further flexibility arises from the use of abstract annotations as the principal representation of intermediate and final results of interprocedural data-flow analysis.

Demand-driven analysis framework. All solutions to interprocedural problems are computed by FIAT on a demand-driven basis. Benefits of this strategy include facilitating modular software development by providing a clean separation between interprocedural problems and eliminating the need for clients to know about the ordering of interprocedural problems.

Parameterized data-flow analysis template. FIAT provides a template for interprocedural data-flow analysis problems. From the programmer's standpoint, using this template minimizes the effort needed to specify a new interprocedural data-flow problem. This feature facilitates rapid prototyping of new interprocedural techniques.

Analysis for interprocedural transformations. FIAT provides a framework suitable to investigate the utility of interprocedural transformations, which modify code based on interprocedural context. Since FIAT deals only with abstract representations of programs, its role in supporting interprocedural transformations is limited to identifying opportunities and assessing the profitability of applying transformations; actual source code transformations are the responsibility of tools that consume FIAT's analysis results.

The next section presents background material on FIAT's architecture, interprocedural analysis and interprocedural transformation. Section 3 describes FIAT's principal representations. Section 4 describes how these representations are used by FIAT's demand-driven analysis engine. Section 5 describes the parameterized data-flow analysis framework and how a user of the system incorporates a new data-flow analysis problem. Section 6 describes existing and planned frameworks for exposing opportunities for interprocedural transformations. Section 7 describes several applications of the system. The paper concludes with a brief discussion of our experiences with FIAT, and our future plans for the system.

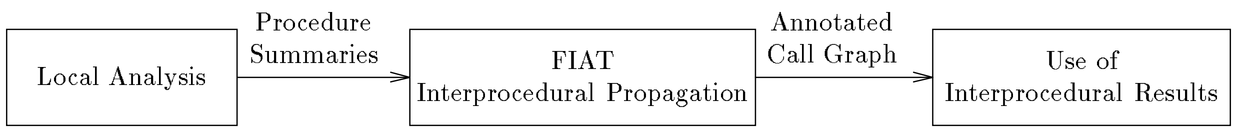

Local Analysis
Procedure Summaries
FIAT Interprocedural Propagation
Annotated Call Graph
Use of Interprocedural Results

Figure 1 A three-phase strategy for whole-program anslysis.

2 Background

To provide some context for the remainder of the paper, here we describe the architecture of whole-program analysis systems for which FIAT was designed and introduce some terminology for interprocedural data-flow analysis and interprocedural transformation.

2.1 Three-Phase Approach to Whole-Program Analysis

FIAT has been designed for use in whole-program analysis systems based on a three-phase strategy [11, 19, 25]:

1. **Local Analysis.** Essential information describing each procedure in a module is collected and saved. This information describes the formal parameters and call sites defining the procedure's interface. In addition, immediate interprocedural effects relevant to particular data-flow problems are collected for use in initializing interprocedural propagation.

2. **Interprocedural Propagation.** FIAT's interprocedural analysis engine examines the local information collected for each procedure and uses its call site information to build a call graph. An iterative data-flow solver then computes solutions to requested interprocedural data-flow problems. Each data-flow problem can use any of the results of local analysis. A data-flow problem can also use results from other data-flow problems as long as dependences between data-flow problems are acyclic. Local analysis results and data-flow solutions may both be used to make decisions about interprocedural transformations.

3. **Use of Interprocedural Results.** Tools that incorporate FIAT's analysis engine can use the results it calculates for display or to direct optimizing transformations of the program.

This three-phase organization supports whole-program analysis while preserving much of the efficiency of separate compilation of procedures. The separate local analysis phase means that there is no need to examine the source code for each procedure during interprocedural propagation. Furthermore, since the initial information for a module does not depend on interprocedural effects, it only needs to be collected once when the module changes rather than every time interprocedural analysis involving that module is performed. This property is true even if the module is used as part of several programs. Using this three phase analysis strategy along with a technique called recompilation analysis, a whole-program compiler could restrict recompilation to only those modules to which interprocedural optimizations were applied that are no longer valid [8].

The FIAT system supports the interprocedural propagation phase. This phase is independent of the intermediate representation of source code used by the other phases. FIAT's input consists of local information for each procedure represented using a system-independent abstraction that is described in the next section. From this information, FIAT computes a call graph annotated with analysis results and interprocedural transformation decisions. Tools inspect interprocedural information available from FIAT's annotated call graph and apply it as appropriate to the tool's purpose. This organization is shown in Figure 1.

2.2 Interprocedural Data-Flow Analysis

Interprocedural data-flow analysis can be described as either *flow insensitive* or *flow sensitive*. Flow-insensitive analysis calculates data-flow effects at program locations without considering the control flow paths that will be taken through individual procedures [4]. In contrast, flow-sensitive analysis derives the effects common to each distinct control-flow path that reaches a location [45].

We use several flow-insensitive data-flow problems as examples throughout the paper:

- MOD(c): the set of variables that may be modified as a result of procedure call c, either by the called procedure or its descendants in the call graph [4].
- REF(c): the set of variables that may be referenced as a result of a procedure call c [4].
- ALIAS(p): the set of variable pairs that may refer to the same memory location in procedure p, resulting from Fortran's call-by-reference parameter-passing mechanism [4].
- CONSTANTS(p): the set of pairs $\langle v, c\rangle$ representing that variable v has constant value c across all calls to procedure p [13].

We use the following example of a flow-sensitive problem in Section 5.

- LIVE(n): the set of variables v that may be used prior to any redefinition [45].

In the above definition, the node n represents either a procedure or a block of code within a procedure.

2.3 Interprocedural Transformations

Figure 2(a) displays a portion of a call graph where procedures `A`, `B` and `C` all invoke procedure `D`. `D` contains three calls to procedures not shown. Interprocedural analysis derives information about the *calling context* for `D` as well as the side-effects of its three call sites, but there are two limitations: (1) Data-flow analysis summarizes information when paths in the call graph merge; and (2) the compiler cannot move code across the call boundary. Interprocedural transformations can sometimes be used to overcome these limitations.

Inline substitution. This transformation replaces a procedure call with a copy of the invoked procedure. In Figure 2(b), by inlining the calls to `D`, each version of `D` may be fully optimized in the context of its caller. However, inlining can sometimes lead to unmanageable code explosion, resulting in prohibitive increases in compile time [47, 18]. Moreover, execution-time performance may degrade when optimizers fail to exploit the additional context inlining exposes or when the code size exceeds register, cache or paging system limits [18, 34, 42, 35, 46].

In light of the limitations of both inline substitution and interprocedural data-flow analysis, our research has explored additional techniques that provide some of the power of inlining but with less associated costs.

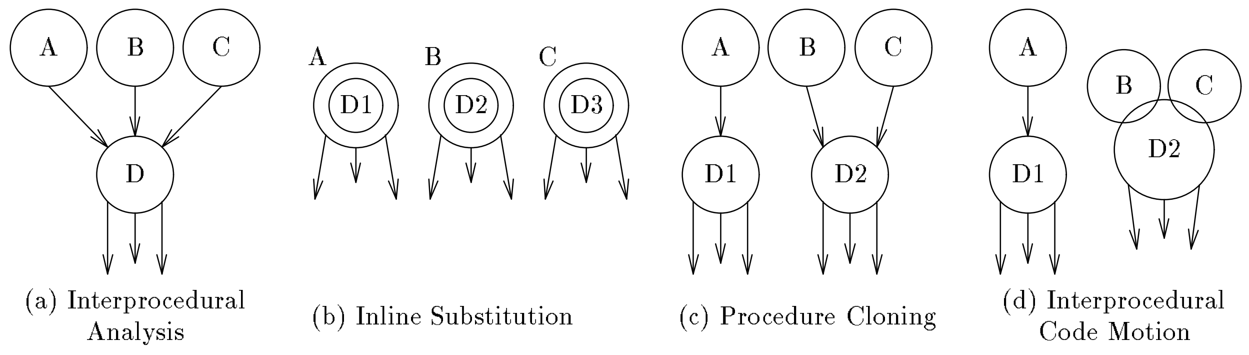


Figure 2 Graphical comparison of interprocedural optimization techniques.

Procedure cloning. To perform this transformation, the compiler replicates a procedure and tailors the procedure and its copy to distinct calling environments [17, 19]. In Figure 2(c), procedure `D` is cloned, inheriting distinct calling context information from `A`. Procedures `B` and `C` both invoke the same copy `D2`. Cloning thus avoids summarization of information inherited from callers. Procedures do not increase in size, and by sharing clones among multiple callers, the compiler can mitigate program growth.

Interprocedural code motion. This transformation moves some portion of code across the procedure boundary. In addition to inline substitution, two other forms of code motion have been explored: *loop embedding* and *loop extraction* [29]. The former moves a loop surrounding a call into the invoked procedure; the latter moves an outer enclosing loop across a call and into the caller. These transformations partially inline the code. However, the compiler controls the code movement; it is thus possible to share transformed procedure bodies via cloning. In Figure 2(d), a small amount of code from procedure `D` has been moved into procedures `B` and `C`, and both procedures invoke the same modified version `D2`.

3 Abstractions

FIAT's analysis engine manipulates three key abstractions: a summary representation of individual procedure bodies, a program call graph, and annotations on the call graph components. Below, we describe each of these abstractions in greater detail.

3.1 Procedure Representation

FIAT's representation of a procedure body is an abstraction that we call `ProcSummary`. Local analysis generates a `ProcSummary` abstraction for each procedure that serves as input to FIAT's analysis engine. A procedure's `ProcSummary` abstraction contains any information about the procedure that will be needed for interprocedural analysis. There are two principal advantages to using this abstraction rather than the procedure body itself. First, the `ProcSummary` abstraction contains only the local facts pertinent to interprocedural analysis and enables FIAT to avoid having the full representation of each procedure body in memory simultaneously during interprocedural analysis. Second, the simplicity of the `ProcSummary` abstraction makes it independent of both source-code language and procedure representation. This property makes FIAT easily retargetable to different systems.

Our current implementation of the `ProcSummary` abstraction, which supports only flow-insensitive analysis, is very simple. The `ProcSummary` representation for a procedure contains identifying information about the procedure, information about the call sites in the procedure, and annotations representing immediate effects for particular data-flow problems. The identifying information for a procedure includes the procedure's name, if it is a function, its return type, and information about its formal parameters including their types, and for array parameters, their dimensions. The information for each call site includes identifying information about the invoked procedure (which may be a procedure variable) and its actual parameters. Annotations for particular data-flow problems represent the immediate effects of a procedure. For example, the immediate effects recorded for the MOD problem include the set of variables that appear on the left hand side of an assignment or are read from a file.

To support flow-sensitive data-flow analysis in FIAT, we plan to extend `ProcSummary` to include a representation of the procedure's control flow graph, with special markings for loop nodes and conditionals. We will augment this graph with special *entry*, *exit*, *call* and *return* nodes, representing procedure entries, procedure exits, call sites and their return points. This additional structure will enable us to record immediate effects at a finer granularity as annotations of nodes in the control flow graph; also, the control flow graph will serve as the basis for propagation of flow-sensitive information in the interprocedural phase. Our early experiences indicate that, at least for Fortran, the control flow graph is much smaller than the code so even our expanded `ProcSummary` representation will remain relatively compact.

3.2 Call Graph

A *call graph* is the principal representation of a program manipulated by FIAT. A call graph contains nodes representing the procedures in the program; edges between nodes represent invocation relationships that arise through call sites in the program. In the simplest case, an edge between nodes representing procedures p and q represents a call site in procedure p that invokes q.

Constructing the call graph. To construct the call graph, FIAT first introduces a node for each procedure in the program. Next, FIAT examines the call site information in the `ProcSummary` representation for each procedure. For each call site that is a direct invocation of a user-defined procedure, FIAT introduces an edge from the node representing the caller to the node representing the callee.

To complete construction of the call graph, FIAT performs *call graph analysis* to determine the set of possible bindings for Fortran-style procedure-valued formal parameters [28]. Call graph analysis simulates parameter passing to compute, for each procedure-valued formal f, the set *Boundto(f)* containing the possible procedure constants that may be passed to f at run time.

To handle calls through procedure-valued formals, FIAT first adds to the call graph a *representative node* for each procedure-valued formal in the program. The system then adds an edge to each formal's representative node from the procedure in which the formal is declared. Finally, for each formal f, the system adds an edge from fs representative node to each procedure b in *Boundto(f)*. During interprocedural analysis, representative nodes serve as a location to summarize interprocedural information from all of the procedures potentially invoked through an invocation of a procedure-valued formal. Summarization is required to correctly report the results of interprocedural analysis, and representative nodes provide a convenient mechanism to obtain this effect without any special accommodations to the data-flow analysis framework or queries on the results of analysis.

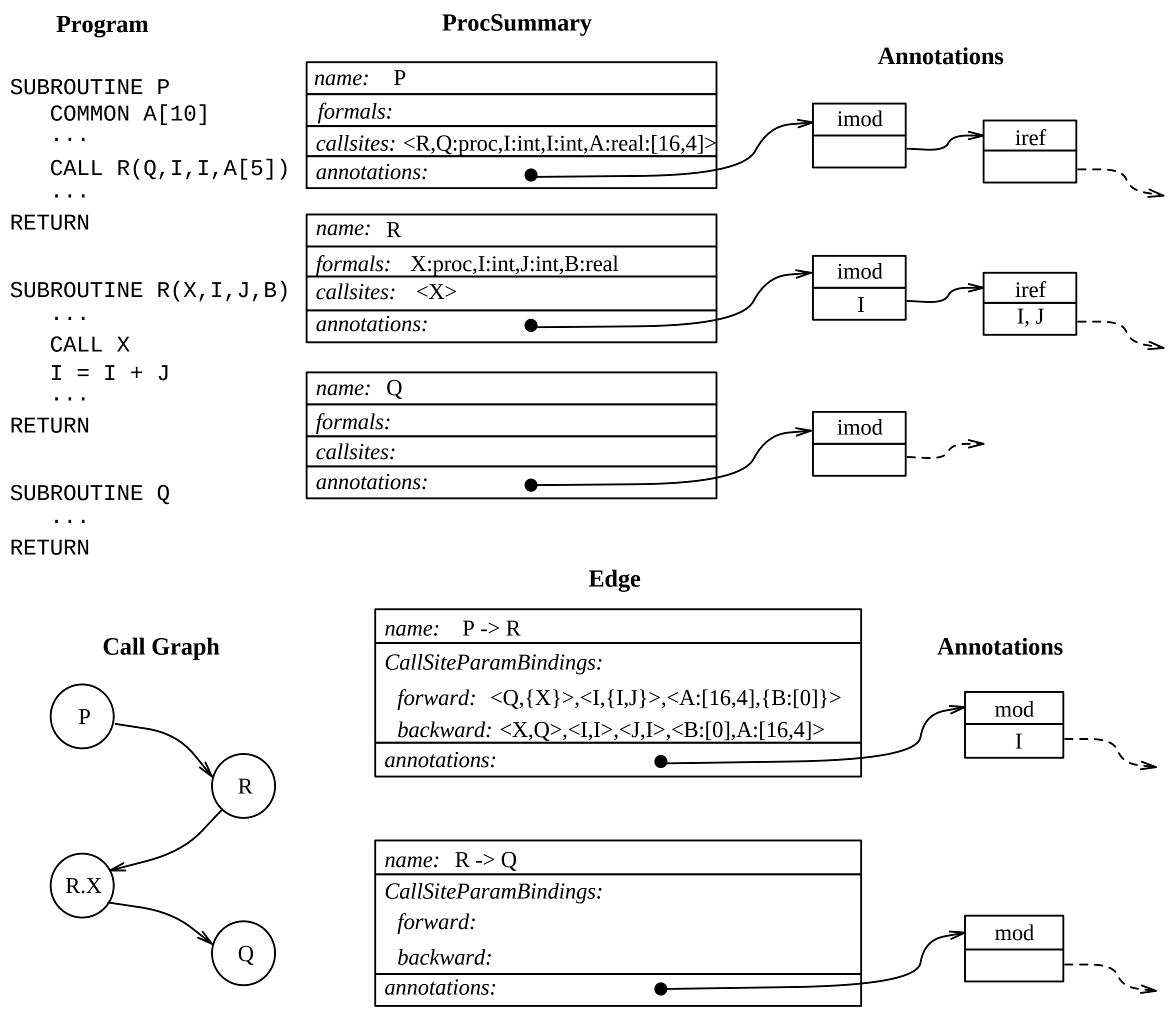
Representation. Each node and edge in the call graph contains a table to hold its associated annotations. Initially, each node in the call graph is annotated with its corresponding `ProcSummary` structure. Each edge in the call graph contains a parameter bindings information structure that details the association of actual parameters with the formal parameters of the invoked procedure. The forward component of the bindings structure provides one-to-many maps relating each variable passed as an actual parameter by the caller to its corresponding formal parameter(s) in the callee. The backward component of the bindings structure provides one-to-one maps from each formal parameter in the callee to its corresponding actual parameter passed at the call. This binding structure, in conjunction with a dictionary of all global variables appearing in the program, is used during data-flow analysis to translate variable names through call sites. During data-flow analysis, both nodes and edges are annotated with additional computed facts.

3.3 Annotations

Annotations are the mechanism for associating arbitrary information with nodes and edges in the call graph. Annotations play an important role in FIAT. Among other things, they are used to represent the intermediate and final results of interprocedural data-flow analysis and to convey interprocedural information from FIAT's driver to tools that use its analysis results.

All annotations are named entities derived from the `CallGraphAnnot` base class. Each annotation must support a basic set of methods for creating, destroying, reading, and writing the annotation. With these methods, the system can manipulate annotations without knowing anything about their internal structure. Annotations that represent data-flow sets must also support a method to test for equality with another annotation; this method is used by FIAT's data-flow solver to determine if the solution to a data-flow problem has converged. Annotations can support additional methods necessary to conveniently manipulate the information they maintain. For example, the ALIAS annotation, which represents the set of potential alias pairs, provides operations for augmenting the set with a new pair of aliases, as well as operations for querying the set of potential aliases for a particular variable.

It is important to note that FIAT completely separates the specification of an annotation representation from the specification of how it is computed. This separation affords opportunities for the reuse of annotation representations. As one important example, a representation of variable sets is shared by a number of interprocedural data-flow problems. The separation also makes it possible to define alternate implementations of the same data-flow analysis problem, for example, to study the use of problem formulations of different precision.



```
SUBROUTINE P
   COMMON A[10]
   ...
   CALL R(Q,I,I,A[5])
   ...
RETURN

SUBROUTINE R(X,I,J,B)
   ...
   CALL X
   I = I + J
   ...
RETURN

SUBROUTINE Q
   ...
RETURN
```

Figure 3 Fiat's abstractions.

3.4 Example

Figure 3 shows an example program with three subroutines, `P`, `Q` and `R`, and their associated representations. The `ProcSummary` abstraction for `P` contains its name and the call to `R`. The information for this call, in the *callsites* field, shows the names and types of the parameters. The tuple $[16, 4]$ indicates the data of length 4 bytes starting at an offset of 16 bytes from the beginning of array `A`. The `ProcSummary` abstraction for `R` is similar, with field *formals* listing the formal parameters and their types. The call site in `R` invokes the procedure parameter `X`. The *annotations* field in `ProcSummary` contains immediate effects for flow-insensitive data flow problems gathered during local analysis. Shown are annotations of the immediate effects needed for Mod and Ref analysis.

The bindings for the call to procedure-valued formal `X` in `R` are resolved during construction of the call graph. In the graph, the representative node `R.X` represents this formal, and the edge from `R.X` to `Q` represents the binding of `X` to `Q`. Each procedure's `ProcSummary` information is associated with its corresponding procedure node in the call graph. Call site information is associated with the corresponding call graph edge. Edge `P` $\rightarrow$ `R` shows in its *CallSiteParamBinding* structure that `I` in `P` is passed to both `I` and `J` in `R`. Annotations on call graph edges and nodes (not shown) represent final solutions to data-flow problems.

4 Interprocedural Analysis Driver

Fiat's interprocedural analysis engine is based upon a demand-driven paradigm. Demand-driven solution of interprocedural problems results in an intuitive interface to the analysis engine that facilitates its incorporation into a tool such as a whole-program compiler. A tool using fiat first initializes fiat's driver to construct a call graph. The tool may obtain the solution to an interprocedural problem at any node (or edge) in the call graph by demanding the annotation corresponding to the problem solution. If the solution has not already been computed, the driver initiates its computation, caches the computed solution so that it is available for future requests, and returns the requested value. Fiat's demand-driven approach has the following desirable properties:

- The solution to any interprocedural data-flow problem can be obtained at any time.
- Explicit ordering constraints between data-flow problems are unnecessary; the only requirement is that there cannot exist a cyclic dependence between problems.
- Implementations of data-flow problems are completely separated from each other and from the solver.

Implementation. To implement the demand-driven paradigm, fiat relies on a set of *annotation managers*, one for each named interprocedural annotation. The distinction between an annotation and an annotation manager is that an annotation embodies a set of information, and the annotation manager embodies a method for computing an annotation. It is useful to separate these two notions since it is often possible to reuse an annotation representation for related problems. For instance we use the same annotation representation for Mod and Ref variable sets.

When one demands by name an annotation on a node (or respectively, edge) in the call graph, the driver inspects the annotation table for the node (edge) for an annotation of the same name. If no such annotation already exists, the driver looks up an annotation manager by that name in a central registry and invokes the manager to compute the requested annotation on the node (edge). For interprocedural data-flow problems, the annotation manager invokes a data-flow solver (described in detail in section 5) to compute the requested annotation.

In the current system, when an interprocedural data-flow solution is demanded for a particular node or edge, the corresponding annotation manager invokes a data-flow solver that computes the solution for the entire call graph. However, the annotation manager framework is suitable for accommodating incremental analysis using a different data-flow solver. Incremental analysis could be used to partially compute solutions on the call graph, such as computing the solution to a forward interprocedural data-flow problem only for the demanded node and its ancestors. Furthermore, the system could incrementally update solutions on demand in response to editing changes in an interactive setting or during recompilation.

To compute annotations that represent information other than the solution of interprocedural data-flow problems, an annotation manager is free to perform whatever actions are necessary. For example, an annotation manager used by the automatic data race detection system built upon fiat (see Section 7) computes the set of variables that might be involved in a data race for each parallel loop. This information is computed by invoking a data dependence analyzer that uses interprocedural Mod information to determine a solution to this problem in the presence of procedure calls.

Ordering computation of data-flow problems. In addition to servicing annotation queries from tools using fiat, the demand-driven organization also implicitly manages the ordering constraints in solving interprocedural problems. If an interprocedural data-flow problem requires the solution of some other problem, it merely demands the results. If they are not yet available, they are computed in response to the demand.

For example, the solution to the Mod problem requires Alias information since a variable is modified if any of its aliases is modified. Thus, the annotation manager computing Mod annotations demands Alias information. If alias information is not already available, the analysis engine is invoked recursively to compute it. Once Alias information is available, it is incorporated into the Mod solution as necessary.

This demand-driven organization ensures that any interprocedural results are available when they are needed without requiring an explicit representation of ordering among data-flow problems. Ordering information is implicitly represented by the code for computing interprocedural problems: a use of the solution to interprocedural problem *p1* in the code for computing interprocedural problem *p2* indicates that *p2* will be solved before computation of *p1* completes. The only restriction a set of interprocedural problems must satisfy for this demand-driven framework to function properly is that there may not be a cyclic dependence among the problems.

Maintaining separation of data-flow problems. The demand-driven organization and the use of a problem registry completely separate the implementations of individual data-flow problems from each other and from the analysis driver itself. The implementation of an annotation manager for a data-flow problem needs only to know which solutions to other data-flow problems it requires.

The separation of interprocedural problems from each other and from the analysis driver ensures that a configuration of FIAT for use with a particular tool includes only those functions related to the problems of interest to the tool. This property is particularly important because FIAT is used for a wide variety of applications (see Section 7). Different systems share some interprocedural problems, such as computation of scalar MOD and REF side effects, but other problems specific to compiling programs for distributed memory multiprocessors or automatically instrumenting shared-memory programs to detect data races are not included when building tools for any other purpose.

5 A Parameterized Template for Interprocedural Analysis

FIAT provides a framework for interprocedural data-flow analysis including a Kildall-style iterative solver [37]. To solve a new interprocedural analysis problem with FIAT, a programmer must provide a local analysis module that collects the immediate effects of a procedure for the new data-flow problem, define an annotation to represent the problem's data-flow sets, instantiate a solver template for the the data-flow problem, and define an annotation manager for the data-flow problem that invokes the provided solver with the template for the problem. Currently, FIAT supports only flow-insensitive problems; however, implementation is under way to support flow-sensitive problems as well.

Below we describe how a programmer specifies a new flow-insensitive data-flow problem in FIAT, how the data-flow solver applies this specification and how we are extending FIAT to support flow-sensitive analysis.

5.1 Flow-Insensitive Analysis

Data-Flow analysis problem template. The programmer provides specifications to be used by the interprocedural data-flow solver to compute values of data-flow sets at nodes and edges in the call graph.

The functions provided to FIAT are simply the specifications for the Kildall data-flow analysis framework [37]:

- `InitNode`(n) and `InitEdge` (e) provide the initial annotations at node n and edge e, respectively. These functions collect information from `ProcSummary` or from the solutions to previous data-flow problems.
- A *meet function* $\bigwedge(s1, s2)$ for a node takes two sets and returns a conservative approximation of both.
- The *transfer function* $\mathcal{T}_c(s)$ translates the elements in set s across call site c to the other endpoint of c, according to parameter passing at the call.[1] This set is met with the initial set at the edge returned by `InitEdge`.
- The *direction* represents whether information is propagated from a node to its descendants (*i.e.,* a forward problem) or from a node to its ancestors (a backward problem).
- $\top$ and $\bot$ sets are used to initialize the entry and exit nodes of the graph.

[1]The transfer function is distinguished from its role in intraprocedural analysis because it maps information across an edge rather than within a node.

FIAT requires an additional function specification from the programmer, called **`FinalizeAnnotation`**, that may incorporate other data before installing the final solution to the data-flow problem. As an example, MOD analysis folds in modifications for aliases of modified variables.

The template is useful for two reasons. First, it succinctly identifies what a programmer must implement to solve a new interprocedural problem with FIAT. Second, the template greatly simplifies integration of new analysis problems into the system and enables functionality common to all data-flow problems to be implemented with shared code.

Iterative solver. FIAT provides a common iterative solver that is shared by all data-flow analysis problems. Abstractly, for forward data-flow problems, the solver calculates the solution to the following simultaneous equations[2]:

$$Set(n) = \bigwedge_{c=p\rightarrow n} \mathcal{T}_c(Set(p))$$

For backward problems the solver similarly calculates the result of the following equations:

$$Set(n) = \bigwedge_{c=n\rightarrow s} \mathcal{T}_c(Set(s))$$

To compute the solution to the set of simultaneous equations representing a data-flow problem, the solver first initializes the data-flow set for each call graph node n to its initial value returned by **`InitNode`**.

$$Set(n) = InitNode(n),$$

and then iterates over the call graph applying the user-defined meet and transfer functions until the solution converges. For forward data-flow problems, the solver iterates over the call graph in reverse post order (in acyclic graphs, this means that each procedure is examined after all of its callers). Backward data-flow problems are processed similarly in post order.

For forward data-flow problems, the desired result is the data-flow set associated with each procedure node in the call graph. For backward problems, the data-flow set for each procedure node is translated to represent the side effects of a particular call site by projecting backwards across each incoming call site edge. Thus, the final solution for a backward problem at a call site is stored on the corresponding call site edge in the call graph.

5.2 Extensions for Flow-Sensitive Analysis

To support flow-sensitive problems, we need to define a modified version of the parameterized data-flow template and iterative solver. Furthermore, local analysis must be augmented to compute a control flow graph for use by the interprocedural driver. Finally, for flow-sensitive problems the immediate effects of a procedure must be associated with nodes in the control flow graph rather than being aggregated at the procedure level as described for flow-insensitive problems in section 3. Below we describe the only changes needed to to the flow-insensitive template and solver to support flow-sensitive analysis.

Data-flow analysis problem template. The programmer must specify two transfer functions. $\mathcal{T}$ maps names from caller to callee. The inverse function $\mathcal{T}^{-1}$ maps in the other direction.

Iterative solver. The solver must now iterate over both the call graph and the control flow graph. The solver begins by initializing sets at all nodes in both the call graph and the control flow graph to $\top$. For a backward problem, we then iterate over the nodes in the call graph until no more changes, performing the following computation for each procedure n:

[2]The meet function actually computes the meet for a pair of sets at a time.

```
(1)     $Set(exit_n) = Set(n)$

        /* Collect information from the entry of called procedures */
(2)     for each call node $c$ in $cfg(n)$
            let $q$ be the procedure invoked at $c$
            $Set(c) = \mathcal{T}_c^{-1}(Set(entry_q))$

        /* Iterate within control flow graph */
        for each node $cn$ in $cfg(n)$
(3)         $Set(cn) = Set(cn) \wedge \bigwedge_{s \in succ(cn)} Set(s)$

        /* Project information to invoked nodes */
(4)     for each return node $r$ in $cfg(n)$
            let $c$ be the corresponding call node
            let $q$ be the procedure invoked at $c$
            $Set(q) = Set(q) \wedge \mathcal{T}_c(Set(r))$
```

This formulation means that Set(n) for some node n representing a procedure contains the set on exit to the procedure. The set value on entry is available, if desired, at the entry node for the procedure.

For forward problems, propagation is similar, but the direction is reversed. Thus, at statement (1), we initialize the set for $entry_n$ rather than $exit_n$. At statement (2), the information at each return node r is initialized from $exit_q$. Statement (3) performs the meet of predecessors, not successors. Statement (4) projects information from the call node rather than the return node. The resulting set at a procedure node n is the value on entry to a procedure; the value on exit is available from the exit node.

5.2.1 An Example

To illustrate how the flow-sensitive solver works, we compute the LIVE sets defined in Section 2.[3] For the live problem, the meet function $\wedge$ is union, and the initial $\top$ set value is $\emptyset$.

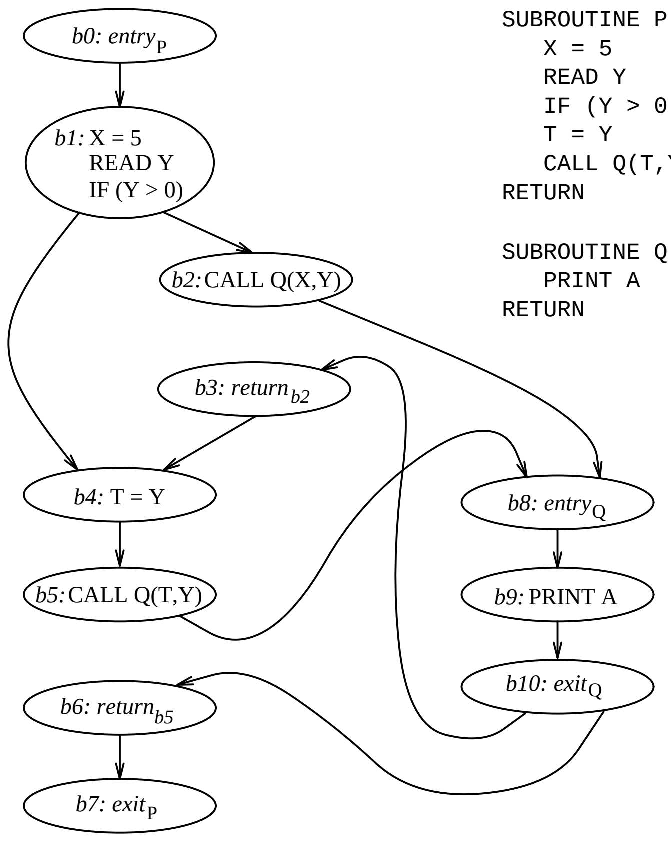
The example program, shown in Figure 4, contains no global variables. We also assume call-by-reference parameter passing semantics. These points allow us to examine the relationship between actual and formal parameters while ignoring the issue of aliasing. The following steps are required to converge at a solution.

	`Initialize`	`Collect`	`Propagate`	`Project`
`Visit P`	Live(b7) = Live(P) = $\emptyset$	Live(b2) = $\mathcal{T}_{b2}^{-1}$(Live(b8)) = $\emptyset$ Live(b5) = $\mathcal{T}_{b5}^{-1}$(Live(b8)) = $\emptyset$ Live(b6) = $\emptyset$	Live(b1) = Live(b0) = $\emptyset$ Live(b4) = Live(b3) = $\{Y\}$	Live(Q) = Live(Q) $\wedge$ $\mathcal{T}_{b5}$(Live(b6)) $\wedge$ $\mathcal{T}_{b2}$(Live(b3)) = $\{B\}$
`Visit Q`	Live(b10) = Live(Q) = $\{B\}$	/* no calls */	Live(b9) = Live(b8) = $\{A, B\}$	/* no calls */
`Visit P`	Live(b7) = Live(P) = $\emptyset$	Live(b2) = $\mathcal{T}_{b2}^{-1}$(Live(b8)) = $\{X, Y\}$ Live(b5) = $\mathcal{T}_{b5}^{-1}$(Live(b8)) = $\{T, Y\}$ Live(b6) = $\emptyset$	Live(b1) = Live(b0) = $\emptyset$ Live(b4) = Live(b3) = $\{Y\}$	Live(Q) = Live(Q) $\wedge$ $\mathcal{T}_{b5}$(Live(b6)) $\wedge$ $\mathcal{T}_{b2}$(Live(b3)) = $\{B\}$

5.2.2 Open Issues.

The flow-sensitive solver described above fails to address two important issues discussed in other research. We outline them here and describe them in detail elsewhere [24]. After experience implementing a number of flow-sensitive data-flow problems, we will consider general solutions to these problems.

[3]At procedure nodes, the solution represents variables live on exit, similar to Callahan's definition [9].



```
SUBROUTINE P
   X = 5
   READ Y
   IF (Y > 0) CALL Q(X,Y)
   T = Y
   CALL Q(T,Y)
RETURN

SUBROUTINE Q(A,B)
   PRINT A
RETURN
```

Figure 4 Example for flow-sensitive LIVE problem.

Unrealizable paths. In our example, we propagated information along paths in the graph that do not correspond to possible execution paths, or *realizable paths* [38]. Specifically, LIVE(b5) = $\{T, Y\}$ is imprecise. It results from propagation of values along the following subpath in the graph: ..., *b4, b3, b10, b9, b8, b5,* .... The body of Q is entered from the return at $b3$ and exits at the other call $b5$.

This situation arises because flow-sensitive analysis propagates information into a procedure describing its *calling context* and information out of a procedure describing its *side effects*. Three basic strategies have been used to avoid propagating information along unrealizable paths: (1) data-flow sets are tagged with path history information that describes the path taken to derive a set [14, 45, 48, 30]; (2) data-flow sets are tagged with some other identifying information related to a specific calling context [38]; or (3) a side-effect computation is performed, ignoring calling context [10]. The third alternative avoids the unrealizable path problem entirely but cannot be applied to every data-flow problem. Tagging with a path history, while general, potentially induces an exponential explosion on the number of data-flow sets (or infinite in the case of recursion) and must therefore be bounded. Moreover, since the tags have no relationship to the data-flow sets, it seems likely that redundancies will arise where sets with distinct tags are identical. Tagging with other identifying data-flow information, the second alternative, seems preferable since it captures information more likely to be useful to enhancing precision.

Sparse program representations. The program representation used in flow-sensitive analysis contains the control flow graph for every procedure, annotated with initial information needed by the propagation phase. This program representation may be quite large, particularly if the annotations on each control flow graph node are nearly as large or larger than the original source code. Moreover, it may be an inefficient representation for data-flow analysis as compared to sparse representations such as the program summary graph [9], the SSA graph [21] or Sparse Data Flow Evaluation Graphs [15]. These graphs provide direct connections in a graph between nodes that create and use data-flow information. The program summary graph and SSA specifically connect definitions of variables with their uses; the def-use relationship is useful for a variety of data-flow problems but is not sufficient as a general approach to data-flow analysis. The sparse data-flow evaluation graph, on the other hand, may be specifically constructed to solve any monotone data-flow analysis problem. However, a system based on the sparse data-flow evaluation graph would require a new graph be constructed for each data-flow problem. The cost of constructing the graph must be weighed with the benefits during propagation.

6 Interprocedural Transformations

This section discusses how a compiler may use FIAT to locate opportunities for profitable interprocedural transformations, as defined in Section 2. When performed in concert with interprocedural data-flow analysis, these transformations expose better information to analysis and enable moving code across procedure boundaries. FIAT *does not* manipulate source code; it supports interprocedural transformations with algorithms that examine annotations on the call graph to locate opportunities where interprocedural transformations are safe and profitable. A compiler may then use this information to perform the mechanical modifications to source code required to complete the transformation.

We present FIAT's existing prototype framework for one of these interprocedural transformations, *procedure cloning*. We then briefly discuss the requirements for frameworks to support other interprocedural transformations.

6.1 A Framework for Procedure Cloning

The design of the cloning framework is based on previous research on how to apply cloning to expose better data-flow information while avoiding the potential for exponential code growth [17]. Two key insights about cloning motivated the implementation:

Refines forward data-flow information. Cloning changes the structure of the call graph in a way that allows interprocedural analysis to proceed along distinct paths. For a forward data-flow problem cloning removes some of the points of confluence — those points where the analysis uses a meet function to approximate the facts that are true along two converging paths. Thus, we can use cloning to directly sharpen the results of forward data-flow analysis problems by replicating nodes with significantly different interprocedural information for different invocations.

Our cloning algorithm may be used to sharpen the results of any forward flow-insensitive interprocedural data-flow problems, such as constant propagation, alias analysis and type analysis. [4] The algorithm further takes advantage of the sharper information at a node by incrementally propagating it to the node's descendants in the call graph.

Goal-directed cloning. The above discussion suggests that we calculate solutions to the forward interprocedural problems and use these directly as the basis for cloning decisions. Unfortunately, compilers cannot capitalize on every new data-flow fact that is exposed. For example, it would not be profitable to clone based on constant values of a variable that is not referenced in the procedure or its descendants. To avoid creating superfluous clones, and thus reduce code growth, we would like to filter the set of interprocedural constants such that the cloner only examines information for referenced variables. Other examples of desirable cloning filters are discussed elsewhere [17, 27].

We define a cloning strategy that filters the data-flow sets to target specific optimization opportunities as *goal-directed* [6]. FIAT's cloning framework supports goal-directed cloning by allowing the programmer to specify a filtering function to distinguish between data-flow facts that have an impact on code quality and those that do not.

[4]Note that the change in the graph may also indirectly sharpen solutions to backward data-flow problems. For example, changes in the results of alias analysis or constant propagation (both forward problems) can change the results of MOD analysis (a backward problem). However, it is unclear how to predict the impact a cloning decision will have on the solution to a backward data-flow problem.

6.1.1 Programmer Specifications

The cloning framework requires minimal specifications beyond specifying the data-flow problem to be used as the basis for cloning. The only additional mandatory specification is the name of the annotation for the result data-flow set at a node or edge. The annotation name is used to partition the incoming edges and to attach the results during incremental updates.

The programmer may optionally specify a `Filter` function to perform goal-directed cloning. For an input data-flow set S at edge e, `Filter`(S, e) returns a set S' which eliminates insignificant data-flow facts from S. For example, to clone based on interprocedural constant propagation, one might specify the following filter:

$$\mathit{Filter}(\textsc{Constants}(e),\, e) = \{\langle v,c\rangle \mid \langle v,c\rangle \in \textsc{Constants}(\mathrm{e}) \textit{ and } v \in \mathrm{G}\textsc{Ref}(\mathit{callee}(e))\}.$$

S' only contains pairs associated with variables globally referenced by the procedure invoked at the edge.

6.1.2 Cloning Algorithm

The algorithm for procedure cloning is given in Figure 5. The algorithm examines the filtered data-flow sets contributed by each incoming edge to a node, partitioning the edges into equivalence classes. Whenever the partitioning results in multiple equivalence classes, an opportunity for cloning exists. Following partitioning of the calls to a node, the algorithm replicates the node and its outgoing edges in the call graph, reassociates the incoming edges to invoke the appropriate cloned version, and incrementally updates the data-flow information at the node and its outgoing edges. The nodes are visited in topological order so that the cloning decision at a node can incorporate the effects of cloning all of its ancestors in the call graph.

For clarity of presentation, this algorithm assumes that the call graph contains no cycles (signifying recursion). Support for recursion is described elsewhere [17]. The algorithm relies on the following two functions:

- The function `SplitNode`(n) replicates node n in the call graph, replicates its outgoing edges, reassociates its incoming edges, and copies its annotations and other associated data.
- The function `InsertPartition`(e, S, Π) creates a new partition $\langle e, S\rangle$ for the collection Π if the data-flow set S does not already appear in the collection. If S is in the collection, it simply adds the edge e to a list representing the call sites to the procedure with the same value for their filtered data-flow set. These call sites will share the same cloned procedure body.

The incremental updates to data-flow sets rely on the transfer function $\mathcal{T}$ and the meet function $\bigwedge$, specified as part of the data-flow problem (see Section 5). The algorithm forms the meet of the sets contributed by the edges in the partition. The resulting set may not be identical to the set associated with the partition since some data-flow facts may have been eliminated by `Filter`, but the differences should not significantly affect optimization. Once the algorithm derives the annotation for a new node, it applies the transfer function $\mathcal{T}$ at each outgoing edge to map variable names across a call, and then incorporates initial data-flow information at the call graph edge. These incremental updates prepare the information for cloning analysis of descendant nodes in the call graph.

6.1.3 Open Issues

Two remaining issues need to be considered for the cloning framework. The first issue deals with incrementally updating the solutions to problems other than the cloning problem. Since the cloning algorithm changes the graph structure, solutions to other forward and backward problems may indirectly become more precise, and the algorithm should also incrementally update these solutions. A similar issue is how to clone based on flow-sensitive data-flow problems. In this case, information resulting from cloning at a node must be propagated through emanating call chains to determine the effects for descendants.

```
for each node $n$ in topological order

    /* Partition filtered data-flow sets contributed by calls to $n$ */
    for each call $c_{in}$ invoking $n$
        $tmpSet = Filter\ (Set(c_{in}))$
        $InsertPartition\ (c_{in}, tmpSet, \Pi(n))$
    endfor

    /* Replicate $n$ in the call graph and incrementally update its data-flow set */
    for each partition $\pi$ in $\Pi(n)$ until threshold for partitions exceeded
        $newNode = SplitNode(\pi)$
        $IncrementalUpdates(newNode, \pi)$
    endfor
endfor


$IncrementalUpdates(node, \pi)$
    /* Node's set is meet of incoming calls in partition */
    $nodeSet = \bigwedge_{c_{in} \in \pi} Set(c_{in})$
    attach $nodeSet$ to $node$
    for each call $c_{out}$ in $n$
        $edgeSet = \mathcal{T}_{c_{out}}\ (nodeSet)$
        attach $edgeSet$ to edge representing $c_{out}$
    endfor
```

Figure 5 Cloning Algorithm.

6.1.4 Examples

We demonstrate the algorithm with two examples. The program for the first example, shown in Figure 6, contains two calls to procedure `P` in `MAIN`, and two calls to procedure `Q` in `P`. The constants sets for the calls to `P` are as follows:

$$\text{Constants}(\text{c1}) = \{\langle I, 1\rangle, \langle G, 20\rangle\}. \quad \text{Constants}(\text{c2}) = \{\langle I, 3\rangle, \langle G, 20\rangle\}.$$

The algorithm begins by filtering these two sets. Since the variable G does not appear in `P` or its descendant `Q`, the algorithm eliminates these elements. Two partitions are created for `P`: $\{\langle I, 1\rangle\}$ and $\{\langle I, 3\rangle\}$. The algorithm replicates `P` in the call graph, associating the first full set (including the pair for G) with `P` and the second with `P'`. It calculates the sets for the four calls to `Q`, using the transfer function to map names across the call and to include the initial pair $\langle N, 10\rangle$ at each call. Sets for the calls to `Q` are the follows.

$$\text{Constants}(c3) = \{\langle N, 10\rangle, \langle INC, 1\rangle, \langle G, 20\rangle\}. \quad \text{Constants}(c4) = \{\langle N, 10\rangle, \langle INC, 1\rangle, \langle G, 20\rangle\}.$$
$$\text{Constants}(c3') = \{\langle N, 10\rangle, \langle INC, 3\rangle, \langle G, 20\rangle\}. \quad \text{Constants}(c4') = \{\langle N, 10\rangle, \langle INC, 1\rangle, \langle G, 20\rangle\}.$$

The algorithm creates two partitions for `Q` and updates the sets, arriving at the final call graph shown in Figure 6. Opportunities for cloning based on interprocedural constants arise often in scientific codes when using multi-purpose, utility or library code. In terms of parallelization, the most common benefit of cloning is in exposing constant loop bounds for parallel loops (and thereby enabling the compiler to directly evaluate whether parallelization is worthwhile). In some cases, cloning simplifies subscript expressions or simplifies control flow in a way that exposes parallelism. The latter situation occurs in the code fragment from the Perfect benchmark SPEC77 shown in Figure 7.

The guard `IFILT` varies on the two calls to `GWATER`. In the second call, which occurs in the main computational loop of the program, both loops shown above in `GWATER` will execute. By eliminating the test on

```
PROGRAM MAIN              SUBROUTINE P (A,I)         SUBROUTINE Q (A,N,INC)
     COMMON G              c3: CALL Q (A,10,I)          ...
     G = 20                c4: CALL Q (A,10,MOD(I,2))
 c1: CALL P(A,1)
 c2: CALL P(A,3)
     PRINT *, G
```

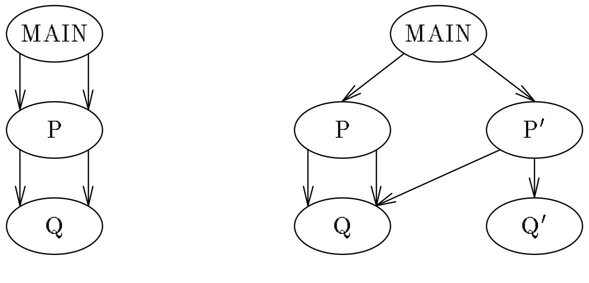


Figure 6 Example for cloning algorithm.

IFILT, the compiler can recognize that fusing these two loop nests is safe. The resulting fused nest contains a larger granularity of parallelism and exposes the reuse of every access to RT(I,K).

6.2 Other Interprocedural Transformations

Similar frameworks are necessary to support other interprocedural transformations such as inline substitution, loop embedding and loop extraction. The SUIF compiler contains a tool that inlines a called procedure into its caller. FIAT could provide a framework to drive this inliner. The inlining framework would walk the edges in the call graph, invoking a decision procedure at each edge to determine whether inlining is desirable. The framework would manipulate the call graph in response to inlining decisions and take care of incremental updates of the data-flow information. A programmer of the framework would need to provide the decision procedure, which would be permitted to examine information in the call graph to make its decision.

Loop embedding and loop extraction require similar support to inlining. Assume that a loop is moved across exactly one call. The system could determine whether embedding and extraction are safe [29]. For instances of safe embedding or extraction, a decision procedure could be invoked to determine if the techniques are desirable. The system would respond by marking the call sites as embedded or extracted, and in the case of embedding, include specifications about the loop index bounds for the embedded loop. The

```
        PROGRAM SPEC77                      SUBROUTINE GWATER(FILTA,DT,IFILT,...)
            ...                                 ...
            CALL GWATER(FILTA,DT,0,...)         IF (IFILT .EQ. 0) GOTO 750
            ...                                 ...
C       MAIN LOOP                               DO K = 1, 7
        DO JDT = LIMLOW, MAXSTP                    DO I = 1, 961
            ...                                       RM(I,K) = CMPLX(FILTA*REAL(RQ(I,K)) +
            CALL GWATER(FILTA,DT,1,...)                  FILTB*(REAL(RT(I,K)) + REAL(RM(I,K))...
                                                         + AIMAG(RT(I,K)) + AIMAG(RM(I,K)))
                                            750 CONTINUE
                                                DO K = 1, 7
                                                   DO I = 1, 961
                                                      RQ(I,K) = RT(I,K)
```

Figure 7 Cloning example from SPEC77.

cloning algorithm could use these marks to make decisions on what callers may share the same optimized procedure body. The data-flow sets would be incrementally updated if necessary.

7 Applications of the System

The current system is being used to solve a variety of data-flow analysis problems across a wide range of applications. The different configurations of the system can be categorized as follows: (1) interactive parallelization for shared-memory multiprocessors; (2) compilation of Fortran D for distributed-memory multiprocessors; (3) static performance estimation; (4) detecting race conditions in programs for shared-memory multiprocessors; and, (5) compilation in the Stanford SUIF compiler. Researchers at Rice and Stanford will use the system to experiment with these configurations. We briefly describe each of these in the paragraphs below.

Interactive parallelization. To safely execute the iterations of a loop containing a call in parallel, the compiler must have knowledge of the side effects of the call. *Dependence analysis*, which provides the compiler's fundamental understanding of a program's inherent parallelism, must be extended to incorporate, in addition to scalar MOD and REF analysis, [5] more precise side-effect information about the subportions of arrays affected by a call [51, 40, 7, 12]. *Regular section analysis*, which derives rectangular descriptions of array accesses due to a call, will serve this purpose [31]. Finally, CONSTANTS and interprocedural symbolic analysis refine dependence information by deriving information about loop bounds and subscript expressions. Of these, MOD and REF are implemented and available in the ParaScope Editor, CONSTANTS is implemented but not integrated, and regular section analysis and interprocedural symbolic analysis are nearly completed.

Static performance estimation. Parallelization and loop transformations are not always profitable if the amount of computation in a loop is insufficient to overcome the synchronization and other overhead of parallelization. Moreover, often when performing loop transformations, the system has a number of possible options that may have substantially different effects on performance. To assist parallelization in making the correct optimization choices, the ParaScope system contains static performance estimation [36]. The performance estimator uses a training set methodology to estimate for a given architecture the execution time of code constructs, in order to compare the effects of different parallelization choices [3]. In order to make decisions about loops containing procedure calls, these performance estimates must be propagated interprocedurally.

Compiling Fortran D. Fortran D is a version of Fortran augmented with data decompositions that specify how data is mapped onto the nodes of a parallel machine [22]. The Fortran D compiler for distributed-memory machines uses data decompositions to partition a program so that a processor only performs computations for data stored locally; it also inserts communication for nonlocal data accessed by each processor. The compiler performs extensive program analysis as well as optimizations such as extracting messages out of loops in order to reduce communication cost [33].

The semantics of Fortran D dictate that a procedure inherits the data decompositions of its formal parameters and global variables from its callers. The compiler must thus propagate decompositions interprocedurally in order to determine how data is partitioned locally. Moreover, an interprocedural approach to communication optimization would enable communication to be extracted out of loops across procedure boundaries. Techniques for interprocedural analysis and optimization of Fortran D are described elsewhere [27]. The current implementation, based on FIAT, contains analysis of decompositions reaching a procedure, and analysis of storage requirements for nonlocal data; interprocedural communication optimizations have not yet been implemented.

[5] It is not clear whether ALIAS information should be used when combining interprocedural information with dependence analysis. Because ALIAS analysis is flow-insensitive, it will falsely report aliases which may lead to overly conservative dependence information [20]. Since it is a violation of the Fortran standard for two variables to be aliased across a procedure call if one of the variables is modified, ignoring aliases is safe for all standard-conforming Fortran programs.

Data race detection. To automatically detect data races during execution of a shared-memory parallel program, the program must be instrumented with calls to a run-time library to monitor individual accesses to variables that might be involved in data races. A conservative instrumentation strategy is to monitor each variable access in the program. However, such a naive instrumentation strategy can cause program execution time to increase by more than a factor of fifteen.

The ERASER data race instrumentation system was developed to identify accesses that could never participate in data races and therefore need not be instrumented and subject to run-time monitoring [43]. Built on top of the ParaScope infrastructure, ERASER uses ParaScope's data dependence analysis to identify pairs of references that might be involved in data races. In the absence of interprocedural information, however, each procedure must assume that it can be called from within a parallel construct and accesses to its formal parameters or global variables must be instrumented. To reduce the number of variable references instrumented and the amount of additional storage required to support data race detection, ERASER relies on the FIAT interprocedural analysis engine to solve three interprocedural problems: determining which formal parameters and global variables need data race instrumentation inside each procedure, which local variables in a procedure require auxiliary storage for use by the race detection run-time library, and which global variables require auxiliary storage.

For the three programs tested with ERASER thus far, using both data dependence analysis and interprocedural analysis reduced the dynamic overhead of instrumentation by 70–100% over the naive strategy. Without interprocedural analysis, data dependence analysis alone provided only modest benefits reducing instrumentation overhead by less than 10%.

Interprocedural optimization in SUIF. The above applications characterize uses of FIAT within ParaScope. Within the Stanford SUIF compiler, we will also use the system to generate code for shared-memory, distributed-memory and distributed shared-memory multiprocessors.

All of these approaches will share a core group of analyses to improve the precision of dependence analysis and detect parallelism. For this purpose, the current system contains MOD and REF analysis and interprocedural constant propagation. In addition, the cloning framework performs cloning based on constant propagation. The remainder of the analysis described above for interactive parallelization in ParaScope will also be incorporated into the SUIF compiler, although the approaches will be different. For example, the approach to interprocedural array privatization will rely on Last Write Trees, which characterize for a given read statement in a loop the iteration corresponding to the last write of that object [41].

The distributed-memory and distributed-shared-memory approaches will perform analysis of reaching decompositions and communication optimization, similar to the techniques used by the Fortran D compiler. However, the SUIF compiler operates on Fortran 77, automatically partitioning data and computation, rather than relying on programmer specifications [1, 2]. This distinction will greatly impact analysis of reaching decompositions, which now requires a flow-sensitive approach. Communication optimization will be based on Last Write Trees.

8 Summary and Future Plans

FIAT provides a framework for interprocedural analysis and transformation that facilitates incorporating interprocedural techniques into existing systems. Parameterized templates for data-flow analysis and procedure cloning and their underlying drivers enable rapid prototyping of new data-flow analysis techniques. System-independent abstractions make FIAT suitable for use in different systems with distinct intermediate representations of programs. Demand-driven analysis maintains a clean separation between data-flow problems, enabling different configurations of data-flow problems to be completely independent.

FIAT has proven useful as a vehicle for experimenting with interprocedural techniques. It has been incorporated into both the ParaScope programming tools and the "D system" programming tools under development at Rice and the SUIF compiler at Stanford without modification to the core system. In addition, new data-flow problems can be added without any modifications to the analysis engine. In a short period of time, a number of external users have developed configurations of the system for a variety of application domains. These different configurations share some data-flow problems, but typically also have problems unique to their domain; the clean separation of data-flow problem implementations means that functions related to problems not part of the current configuration are neither invoked nor linked into the system.

We anticipate a number of extensions to this system. Our current efforts are focused on completing the flow-sensitive analysis support. We plan to use the system to solve a number of flow-sensitive data-flow problems and derive features of these problems that will ultimately lead to efficient, general support. We are also actively developing a better general-purpose interface for specifying local analysis as part of the FIAT system. In the future, we will fully incorporate frameworks for interprocedural transformations other than procedure cloning, particularly inline substitution.

Acknowledgements. The authors wish to thank Jennifer Anderson, Monica Lam, Brian Murphy and Chau-Wen Tseng for their insightful comments on this paper and John Hennessy, Ken Kennedy and Monica Lam for encouraging this work.

References

[1] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed-memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 28(7). ACM, July 1993.

[2] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 28(7). ACM, July 1993.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* Williamsburg, VA, Apr. 1991.

[4] Banning, J.P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*, pages 29–41. ACM, Jan. 1979.

[5] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, Nov. 1992.

[6] Briggs, P., Cooper, K.D., Hall, M.W., and Torczon, L. Goal-directed interprocedural optimization. Technical Report TR90-148, Dept. of Computer Science, Rice University, Nov. 1990.

[7] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pages 163–275. ACM, July 1986.

[8] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, to appear 1993.

[9] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[10] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 47–56. ACM, July 1988.

[11] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[12] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[13] Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[14] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*. ACM, Jan. 1993.

[15] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 55–66, Orlando, FL, Jan. 1991.

[16] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, Feb. 1993.

[17] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, Apr. 1992.

[18] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[19] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, Oct. 1986.

[20] K. D. Cooper, M. Hall, and L. Torczon. Unexpected side effects of inline substitution – a case study. *Letters on Programming Languages and Systems*, 1(1), Mar. 1992.

[21] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages*, June 1989.

[22] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, Dec. 1990.

[23] D. Grove. Interprocedural constant propagation: a study of jump function implementations. Master's thesis, Dept. of Computer Science, Rice University, Mar. 1993.

[24] M. Hall. Comparing FIAT's support of flow-sensitive interprocedural data-flow analysis with existing techniques. Technical Note Available from First Author. To be published as a Stanford Dept. of Computer Science Technical Report.

[25] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Dept. of Computer Science, Rice University, Apr. 1991.

[26] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the ParaScope Editor: an interactive parallel programming tool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[27] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, Nov. 1992.

[28] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3), Sept. 1992.

[29] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[30] W. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, Oct. 1989.

[31] Havlak, P. and Kennedy, K. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[32] M. Hind, M. Burke, P. Carini, and S. Midkiff. Interprocedural array analysis: How much precision do we need? In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[33] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory

machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[34] A. M. Holler. *A Study of the Effects of Subprogram Inlining.* PhD thesis, Univ. of Virginia, Charlottesville, VA, May 1991.

[35] W. W. Hwu and P. P. Chang. Inline function expansion for inlining C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation.* ACM, June 1989.

[36] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation in a parallelizing compiler. Technical Report TR91-174, Dept. of Computer Science, Rice University, Dec. 1991.

[37] Kildall, G. A unified approach to global program optimization. In *Conference Record of the Symposium on Principles of Programming Languages*, pages 194–206. ACM, Jan. 1973.

[38] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 27(7), pages 235–248, July 1992.

[39] Z. Li and P.-C. Yew. Interprocedural analysis and program restructuring for parallel programs. Technical Report CSRD-720, University of Illinois, Urbana-Champaign, Jan. 1988.

[40] Li, Z. and Yew, P. Efficient interprocedural analysis for program restructuring for parallel programs. *ACM SIGPLAN Notices*, 23(9):85–99, 1988.

[41] D. Maydan, S. Amarasinghe, and M. Lam. Array data-flow analysis and its use in array privatization. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages.* ACM, Jan. 1993.

[42] S. McFarling. Procedure merging with instruction caches. In *SIGPLAN '91 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 26(6), pages 71–79, June 1991.

[43] J. M. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, May 1993.

[44] Metzger, R. and Stroud, S. Interprocedural constant propagation: an empirical study. *ACM Letters on Programming Languages and Systems*, 1(2), June 1992.

[45] E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual Symposium on Principles of Programming Languages.* ACM, Jan. 1981.

[46] S. Richardson. *Evaluating interprocedural code optimization techniques.* PhD thesis, Stanford University, Dept. of Electrical Engineering and Computer Science, Stanford, CA, Feb. 1991.

[47] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, Aug. 1989.

[48] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice Hall Inc, 1981.

[49] J. Singh and J. Hennessy. An empirical investigation of the effectiveness of and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, Apr. 1991.

[50] S. Tjiang, M. E. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, Aug. 1991. Springer-Verlag.

[51] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.