Interprocedural Dataflow Analysis in the Presence of Large Libraries

Atanas Rountev[1], Scott Kagan[1], and Thomas Marlowe[2]

[1] Ohio State University, Columbus, OH, USA
[2] Seton Hall University, South Orange, NJ, USA

Abstract. Interprocedural dataflow analysis has a large number of uses for software optimization, maintenance, testing, and verification. For software built with reusable components, the traditional approaches for *whole-program analysis* cannot be used directly. This paper considers *component-level analysis* of a main component which is built on top of a pre-existing library component. We propose an approach for computing summary information for the library and for using it to analyze the main component. The approach defines a general theoretical framework for dataflow analysis of programs built with large extensible library components, using pre-computed summary functions for library-local execution paths. Our experimental results indicate that the cost of component-level analysis could be substantially lower than the cost of the corresponding whole-program analysis, without any loss of precision. These results present a promising step towards practical analysis techniques for large-scale software systems built with reusable components.

1 Introduction

Interprocedural dataflow analysis is a widely-used form of static program analysis. Dataflow analysis techniques play an important role in tools for performance optimization, program understanding and maintenance, software testing, and verification of program properties. Unfortunately, the use of interprocedural dataflow analysis in real-world software tools is hindered by several serious challenges. One of the central problems is the underlying analysis model implicit in most of the work in this area. The key feature of this model is the assumption of a *whole-program analysis for a homogeneous program.* Interprocedural whole-program analysis takes as input an entire program and produces information about the behavior of that program. This classical dataflow analysis model [28] assumes that the source code for the whole program is available for analysis.

Modern software presents serious challenges for this traditional model. For example, systems often contain reusable components. Whole-program analysis assumes that it is appropriate to analyze the source code of the entire program as a single unit. However, for software built with reusable components,

– Some program components may be available only in binary form, without source code, which makes whole-program analysis impossible.

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 2–16, 2006.
© Springer-Verlag Berlin Heidelberg 2006

- It is necessary to re-analyze a component every time this component is used as part of a new system. For example, a library may be used in many applications, and whole-program analysis requires re-analysis of this library from scratch in the context of each such application.
- Code changes in one component typically require complete re-analysis of the entire application.
- The cost of whole-program analysis is often dominated by the analysis of the underlying large library components (e.g., standard libraries, middleware, frameworks, etc.). To achieve practical cost, analysis designers are often forced to use semantic approximations that reduce the precision and usefulness of the analysis solution.

These issues limit the usefulness of many existing analyses. In some cases the analyses cannot be used at all. Even if they are possible, the analyses have to be relatively approximate in order to scale for large-scale software with hundreds of thousands (or even millions) lines of code. Such approximations lead to under-optimized code in optimizing compilers, spurious dependencies in program understanding tools, false warnings in verification tools, and infeasible coverage requirements in testing tools.

Component-Level Dataflow Analysis. In this paper we consider a model of interprocedural dataflow analysis which we refer to as *component-level analysis* (CLA). A component-level analysis processes the source code of a single program component, given some information about the environment of this component. The general CLA model is discussed in [20] (without any formalisms, proofs, or experiments.) Here, we focus on one particular scenario for CLA: analysis of a main component *Main* which is built on top of a library component *Lib*. In this scenario, the source code of *Lib* is pre-analyzed independently of any library clients. This pre-analysis produces *summary information* for *Lib*. This information is used subsequently for component-level analysis of the source code of any main component built on top of *Lib*.

This form of CLA has significant real-world relevance. In particular, there are large standard libraries that are associated with languages such as C++, Java, and C#. A library could be considered as component *Lib*, while a program written on top of it is component *Main*. CLA allows (1) analysis of *Main* without the source code of *Lib*, by using the summary information, (2) reduction in the cost of analyzing *Main*, because the source code of *Lib* has already been analyzed, (3) reuse of the summary information across multiple main components, in order to avoid repeated re-analysis of *Lib*, and (4) reduced work to handle code changes, since changes in *Main* do not require re-analysis of *Lib*.

Contributions. The main goal of our work is to define general theoretical machinery for designing component-level analyses of *Main*. We achieve this goal by generalizing the "functional approach" to whole-program analysis due to Sharir and Pnueli [28]. The key technical issue that this generalization needs to address is *the lack of complete call graph information when performing pre-analysis of a library.* An example of this problem is the presence of callbacks from the library to the main component. The contributions of our work are:

- **General theoretical framework:** This paper defines a general approach for component-level analysis in the absence of complete information about calling relationships within *Lib* and from *Lib* to *Main*. The approach is defined for the most general category of monotone dataflow problems. As a result, it becomes possible to define CLA versions for many important and widely-used whole-program analyses.
- **Framework instantiation:** Our long-term goal is to design CLA versions of existing whole-program analyses, based on the framework from above. In this paper, we show how to instantiate the general approach to a particular form of the *interprocedural reaching definitions analysis*, which is a classical dataflow problem. This analysis exemplifies the category of flow- and context-sensitive dataflow analyses, which present the most challenging targets for our theoretical approach.
- **Experimental comparison:** We present an experimental study which compares CLA with its whole-program counterpart. The experiments indicate that the CLA approach can produce significant reduction in analysis cost, while at the same time achieving exactly the same precision.

2 Whole-Program Analysis

This section describes, at a high level, the classical formulation of whole-program interprocedural dataflow analysis [28]. The input to the analysis is the source code for a complete program. One of the procedures[1] is designated as the main procedure *main*. In the traditional model presented below, each call site invokes only one procedure. A call that could invoke many procedures (e.g., due to virtual dispatch or function pointers) can be modeled as a case statement where each case corresponds to one unique target procedure. Given a complete program, a whole-program analysis constructs a tuple $\langle G, L, F, M, \eta \rangle$ where

- $G = (N, E)$ is an interprocedural control-flow graph (ICFG).
- L is a meet semi-lattice, with partial order $\leq$, meet operation $\wedge$, and greatest element $\top$. To simplify the discussion, we assume that L has finite height.
- $F \subseteq \{f \mid f : L \rightarrow L\}$ is a monotone[2] function space that is closed under functional composition and functional meet.
- $M : E \rightarrow F$ is an assignment of dataflow functions to graph edges. Function $f_e = M(e)$ encodes the effects of e's execution.
- $\eta \in L$ is the solution at the start node of *main*.

Graph G contains the control-flow graphs (CFGs) for the individual procedures. Nodes $n \in N$ correspond to statements, and intraprocedural edges $e \in E$ represent flow of control within the same procedure. The CFG for a procedure p has an artificial start node $start_p$ and an artificial exit node $exit_p$. Each single-target call is represented by two nodes: a *call-site* node and a *return-site* node.

[1] We will use "procedure" to refer to both procedures and methods.
[2] That is, $x \leq y$ implies $f(x) \leq f(y)$ for any $f \in F$ and $x, y \in L$.

There is an interprocedural edge $e \in E$ from a call-site node to the start node of the invoked procedure p; there is also a corresponding edge $e \in E$ from $exit_p$ to the return-site node. Dataflow functions are associated with these edges to represent the effects of parameter passing and return values.

A path in G is a sequence of edges $q = (e_1, \ldots, e_k)$ such that the target of e_i is the same as the source of e_{i+1}. The dataflow function associated with q is the composition of the edge functions: $f_q = f_{e_k} \circ \ldots \circ f_{e_1}$. Not all ICFG paths represent possible executions. A *valid* path has interprocedural edges that are properly matched: each (exit,return-site) edge is matched correctly with the last unmatched (call-site,start) edge on the path.

The *meet-over-all-valid-paths solution* MVP_n for an ICFG node n describes the program properties immediately before the execution of n. This solution is $MVP_n = \bigwedge_{q \in VP(n)} f_q(\eta)$ where $VP(n)$ is the set of all valid paths q leading from the start node of *main* to n (paths q do not include n itself). An analysis algorithm computes a solution $S_n \in L$ at each node n; this solution is *safe* (i.e., correct) if $S_n \leq MVP_n$. There are well-known general algorithms for computing safe solutions for dataflow problems; one such algorithm is outlined in Section 2.2.

2.1 Running Example

We will use the example in Figure 1 throughout the rest of the paper; the figure also shows the corresponding ICFG. The example uses a C-style language to illustrate a whole program built with two components: a library component and a main component. We consider the classical *reaching definitions* problem. The definitions `k=0`, `k=2`, `k=3`, `k=7`, and `k=9` will be denoted by d_0, d_2, d_3, d_7, and

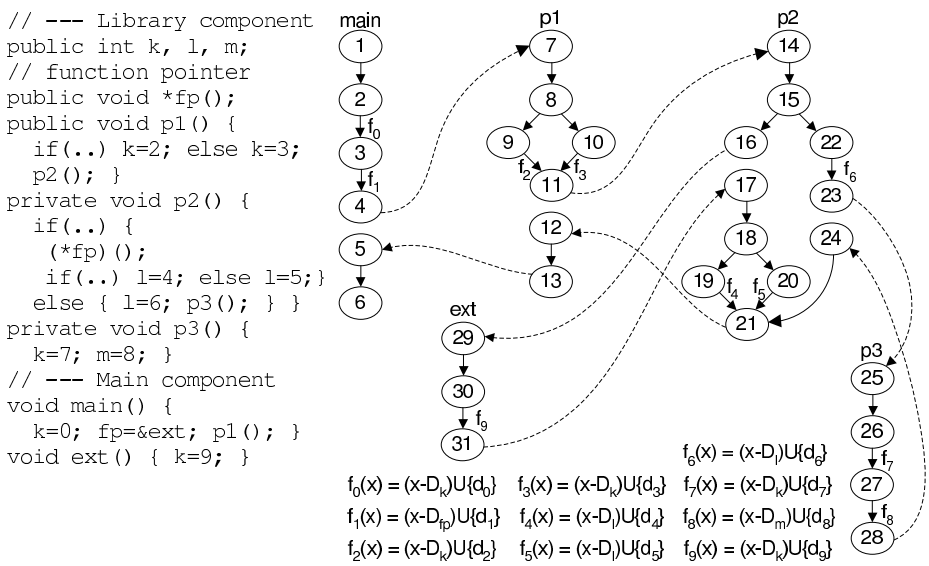

Fig. 1. Whole program, ICFG, and dataflow functions

d_9 respectively. The set of all definitions of `k` will be denoted by D_k. Similar notation will be used for the remaining variables. The lattice for the problem is the powerset of $D = \{d_0, \ldots, d_9\}$, with partial order $\supseteq$, meet operation $\cup$, top element $\top = \emptyset$, and bottom element $\bot = D$. The non-identity dataflow functions are shown next to the corresponding edges in Figure 1. For example, the function for `k=3` is $f_3(x) = (x - D_k) \cup \{d_3\}$, where $x \subseteq D$.

2.2 The Functional Approach of Sharir and Pnueli

One of the classical techniques for solving whole-program dataflow problems is the "functional approach" by Sharir and Pnueli [28]. The essence of this approach is the creation and use of *summary functions*. A summary function $\phi_n : L \to L$ for a node n represents the solution at n as a function of the solution at the start node of the procedure containing n. For example, in Figure 1, $\phi_{27} = f_7$, $\phi_{28} = f_8 \circ f_7$, and $\phi_{11} = f_2 \wedge f_3$. (As usual, for any $g, h : L \to L$, the functional meet $k = g \wedge h$ is such that $k(x) = g(x) \wedge h(x)$ for any x.) In the case when n is the exit node of a procedure p, ϕ_n can be used as a summary function f_p for the entire procedure.

Phase I of Sharir-Pnueli's analysis computes summary functions for all ICFG nodes. This fixed-point computation uses the summary functions for p's callees to compute the summary function for nodes inside p. For example, in Figure 1, $f_{p3} = f_8 \circ f_7$ and $f_{ext} = f_9$. Inside `p2`, these functions can be used to compute, for example, $\phi_{19} = f_{ext}$ and $\phi_{21} = (f_4 \circ f_{ext}) \wedge (f_5 \circ f_{ext}) \wedge (f_{p3} \circ f_6)$. In the output of the first phase, we have $f_{p3} = \phi_{28} = f_8 \circ f_7$, $f_{ext} = \phi_{31} = f_9$, $f_{p2} = \phi_{21}$, $f_{p1} = \phi_{13} = (f_{p2} \circ f_2) \wedge (f_{p2} \circ f_3)$, and $f_{main} = \phi_6 = f_{p1} \circ f_1 \circ f_0$.

Phase II of the analysis propagates lattice elements using the summary functions. In Figure 1, the value $\eta = \emptyset$ at the start node 1 of `main` is propagated to call-site node 4, as $\phi_4(\eta) = (f_1 \circ f_0)(\emptyset) = \{d_0, d_1\}$. This value is then propagated to the start node of `p1`, and from there to call-site node 11 as $\phi_{11}(\phi_4(\eta))$. In turn, the value at 11 is propagated to the start node of `p2`, and to call-site nodes 16 and 22 as $\phi_{16}(\phi_{11}(\phi_4(\eta)))$ and $\phi_{22}(\phi_{11}(\phi_4(\eta)))$, respectively. In general, the propagation occurs only among start nodes and call-site nodes, and stabilizes when the solutions at start nodes are fully computed. Phase III of the analysis can be performed on demand. Whenever the solution at a node n is needed, it can be computed as $\phi_n(S_{start})$, where S_{start} is the solution computed by phase II for the start node of n's procedure.

2.3 Using the Functional Approach for Component-Level Analysis

In Sharir-Pnueli's approach the bulk of the computation is performed during phase I when all ICFG nodes need to be visited, possibly multiple times. Phase II involves only start nodes and call-site nodes. Phase III is performed on demand, and its cost is proportional to the number of distinct queries made by an analysis client. In this paper we focus our efforts on reducing the cost of phase I by using pre-analysis of *Lib*.

In the simplest case, the pre-analysis of *Lib* and the subsequent component-level analysis of *Main* are trivial. Suppose each call site in *Lib* is monomorphic

and its target does not depend on the code in the main component. For example, this is true for C programs that do not use function pointers, and for Java programs that do not make virtual calls. Clearly, the phase I computation for all library procedures can be performed independently of any main component. The summary functions f_p for all exported library procedures p can be stored as the library summary. Later, when CLA of some main component is performed, the phase I computation for that *Main* will compute ϕ_n for all nodes n in this main component, using the pre-computed summary functions for library procedures. Phase II can be restricted only to the portion of the call graph that is in *Main*, and phase III can answer on-demand questions about the node solutions in *Main*.

Unfortunately, this approach is possible only in the *absence of callbacks* from *Lib* to *Main*. However, callbacks are common in real-world software. For example, in C code, one of the parameters of a library function `p` could be a function pointer `g` to a callback function defined in *Main*. A call `(*g)(..)` inside `p` invokes the callback function. Clearly, the complete behavior of `p` is not known at summary-generation time, and it is not possible to create a summary function f_p. This is a realistic problem, because callbacks through function pointers occur often in C libraries [17]. Callbacks also occur often in object-oriented software. Consider a library method `m(A *a)` in C++ or `m(A a)` in Java, where `A` is a library class. Suppose some *Main* creates a subclass `B` of `A` that overrides some of `A`'s methods. If *Main* calls `m` with an actual parameter that is a pointer to an instance of `B`, a virtual call through `a` inside `m` may invoke a method defined in `B`. Of course, this situation is common for extensible object-oriented libraries.

Even in the absence of callbacks, in many cases it is still not possible to create a precise summary function for a library procedure. Consider the following Java example: the library contains a procedure `p` with a virtual call `a.m()`, where the static type of `a` is library class `A`. Suppose the library contains classes `B` and `C` that are subclasses of `A`, and method `A.m` is overridden by `B.m` and `C.m`. A conservative pre-analysis of the library has to assume that `a.m()` could invoke any of these three methods, and as a result the summary function for `p` will depend on all three callees. But if some *Main* instantiates and uses only `C`, for example, the pre-computed summary function for `p` will be overly conservative. Since the calling relationships at virtual calls in *Lib* depend on the execution context created by *Main*, any library procedure that contains polymorphic calls presents a problem for the functional approach.

3 Summary Computation for Component-Level Analysis

Consider again the program in Figure 1. Suppose the library component *Lib* were built as a reusable component, independent of the particular main component in the figure. Furthermore, for the sake of the example, assume that `p1` were made visible to (and callable by) future main components, while the remaining library procedures were hidden from such components using some language mechanism.

Suppose we wanted to compute summary information for *Lib* in order to use it later when performing component-level analysis of any main component *Main* (including the main component from Figure 1). The summary will be main-component-independent, and the only information used for computing this summary will be the source code of *Lib*. Our goal is to construct a library summary with the following property: the subsequent summary-based CLA of *Main* must produce for each ICFG node n in the main component *the same solution* as the solution that would have been computed for n by a whole-program analysis of the source code of $Main \cup Lib$.

One possible summary information contains the ICFG of the library together with some encoding of the dataflow functions at the ICFG edges. However, such a summary contains redundant details that are irrelevant for the CLA of *Main*. As a result, phase I of CLA will have *the same cost* as phase I of a whole-program analysis would have had. Furthermore, due to the redundant information, the summary would be unnecessarily large, making it expensive to store and read.

We propose a general approach that can be used to create a more concise library summary. The approach will be illustrated for the example in Figure 2, but this technique is conceptually applicable to any interprocedural dataflow analysis. (In [22] we show how to handle flow- and context-insensitive analyses.) The basic idea is to compute summary functions that capture the effects of all relevant *library-local ICFG paths*. The functions are then included in the library summary together with information about the program points that could be affected by future main components. Figure 2 shows the summary information computed for *Lib* by our approach. Combining this summary with the ICFG for any main component *Main* allows component-level analysis of *Main*.

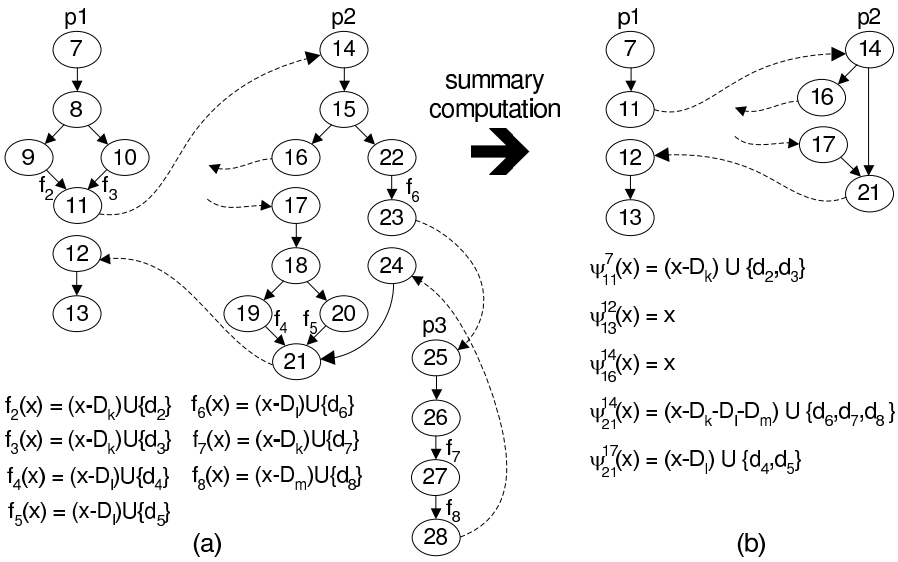

Fig. 2. (a) ICFG for *Lib* (b) condensed ICFG and summary functions for *Lib*

3.1 Library Pre-analysis for Summary Generation

Fixed calls. The pre-analysis of the source code of *Lib* constructs the ICFG for the library and identifies the calls at which the target procedures may depend on the main component. If a call site in *Lib* always invokes the same library procedure, regardless of the code in any main component, we will refer to it as a *fixed call*. In C code, any call that is not through a function pointer is a fixed call. In Java, we can use the following simple criterion for fixed calls: (1) any call that does not correspond to the bytecode instructions `virtualinvoke` or `interfaceinvoke` is a fixed call, and (2) a `virtualinvoke` call is fixed if the static type of the receiver is a final class, or the compile-time target method is a final or private method. Note that it may be possible to employ conservative analyses (e.g., [23, 13]) to identify additional fixed calls that do not satisfy these rather restrictive criteria; such analyses can be performed with worst-case assumptions about the behavior of the unknown main components.

Fixed procedures. We will recursively define a library procedure p to be *fixed* if (1) p contains no calls, or (2) p contains only fixed calls and they invoke only fixed procedures. All transitive callees of p are known at library pre-analysis time, and the Sharir-Pnueli approach can be used to compute a summary function f_p. The library pre-analysis identifies all fixed procedures p and computes the corresponding summary functions. In Figure 2 the only fixed procedure is `p3` and the analysis computes $f_{p3} = f_8 \circ f_7$.

Non-fixed procedures. After processing the fixed procedures, the library pre-analysis considers all non-fixed procedures. The analysis computes a set of summary functions $\psi_n^k : L \rightarrow L$ for each ICFG node n in each such procedure. Here k is an ICFG node that belongs to the same procedure as n, and is one of the following: (1) the start node of the procedure, (2) the return-site node for a non-fixed call, or (3) the return-site node for a fixed call to a non-fixed procedure. Intuitively, k represents a program point which depends on unknown main components. For example, for node 21 in Figure 2, the approach would construct two functions ψ_{21}^{14} and ψ_{21}^{17}. During phase I of the subsequent component-level analysis of a main component, these functions allow us to express the summary function at 21 as $\phi_{21} = \psi_{21}^{14} \wedge (\psi_{21}^{17} \circ f_{callback} \circ \psi_{16}^{14})$, where $f_{callback}$ is the summary function for the callback procedure from the main component.

Computation of summary functions. Figure 3 defines a worklist-based algorithm for computing the summary functions. The algorithm first initializes functions ψ_k^k to the identity function $\lambda x.x$. For all other n, ψ_n^k is initialized to a function that maps every lattice element to $\top$; as usual, $\top$ represents the lack of any information. In our running example, the identity function is associated with nodes 7, 12, 14, and 17. After initialization, functions ψ_n^k are computed incrementally using functional composition and functional meet. A function ψ_n^k captures the semantic effects of certain ICFG paths from k to n. For example, there are two paths from node 7 to node 11 in Figure 1. Function f_2 is propagated to 11 along

procedure *Summary_Computation_For_Nonfixed_Procedures*
for each non-fixed procedure p **do**
  initialize $\psi^{start_p}_{start_p} := \lambda x.x$ and put $(start_p, start_p)$ on the worklist
for each return-site r for a non-fixed call, or for a fixed call to a non-fixed procedure **do**
  initialize $\psi^r_r := \lambda x.x$ and put (r, r) on the worklist
for each other node n and each applicable k **do** initialize $\psi^k_n := \lambda x.\top$
while the worklist is not empty **do**
  remove pair (k, n) from the worklist
  case 1: if n is not a call-site node or a method exit **do**
    for each ICFG successor node n' of n **do** $propagate(k, n', f_{(n,n')} \circ \psi^k_n)$
  case 2: if n is a fixed call-site, with return-site r, calling fixed procedure p **do**
    $propagate(k, r, f_{(exit_p,r)} \circ f_p \circ f_{(n,start_p)} \circ \psi^k_n)$
  case 3: in all other cases, do nothing
procedure $propagate(k, n, f)$
  $\psi^k_n := \psi^k_n \wedge f$
  if ψ^k_n has changed, put (k, n) on the worklist

Fig. 3. Computation of summary functions for non-fixed library procedures

one path, and function f_3 is propagated along the other path. Thus, the summary analysis will compute $\psi^7_{11} = f_2 \wedge f_3 = \lambda x.(x - D_k) \cup \{d_2, d_3\}$. As another example, at return-site 24 we have $\psi^{14}_{24} = f_{p3} \circ \psi^{14}_{23} = f_8 \circ f_7 \circ f_6$.

The computed summary functions are then used to construct the library summary. First, for every fixed procedure p that is visible to future main components, the summary includes the summary function f_p. For every non-fixed p, the summary contains the set Ψ_p of all functions ψ^k_n such that n is (1) the exit node of p, (2) the call-site node for a non-fixed call in p, or (3) the call-site node for a fixed call in p to a non-fixed procedure. For example, for `p1` in Figure 2, $\Psi_{p1} = \{\psi^7_{11}, \psi^{12}_{13}\}$ because 11 is a call-site node for a fixed call to the non-fixed procedure `p2`. Similarly, for `p2`, the summary contains $\Psi_{p2} = \{\psi^{14}_{21}, \psi^{14}_{16}, \psi^{17}_{21}\}$.

The functions in Ψ_p implicitly define a "condensed" CFG for p. The nodes in this condensed graph are all k and n such that $\psi^k_n \in \Psi_p$. For every $\psi^k_n \in \Psi_p$ that is different from $\lambda x.\top$, there is an edge from k to n in the condensed CFG, with edge dataflow function ψ^k_n. These edges represent sets of paths from the original CFG. Figure 2(b) shows the condensed graphs for non-fixed procedures `p1` and `p2`. The condensed CFG for `p3` (not shown in the figure) has only a start node, an exit node, and a single edge with edge dataflow function f_{p3}.

Note that the summary functions are being constructed without any knowledge about the future main components and about the lattice elements that correspond to these main components. For the running example, the summary analysis has no information about the definitions that are generated by main components. For example, in $\psi^7_{11} = \lambda x.(x - D_k) \cup \{d_2, d_3\}$, the set D_k of definitions of `k` is not known completely. However, complete knowledge of D_k is not necessary to encode this function. It is enough to represent the fact that *all* definitions of `k` are killed—both the known ones from the library and the unknown ones created by future main clients.

3.2 Analysis of a Main Component

The summary functions and the condensed CFGs defined by them can be used to perform component-level analysis of any main component that is built on top of *Lib*. Such an analysis is straightforward. First, the condensed CFGs for the library procedures are added to the ICFG for the main component, together with the appropriate interprocedural edges. The resulting condensed ICFG is used as input to phase I of Sharir-Pnueli's whole-program analysis. For any node n in the condensed ICFG, the summary function ϕ'_n computed by this phase I is a safe approximation of the summary function ϕ_n that would have been computed for n by phase I of the standard whole-program analysis of the "normal" non-condensed ICFG—in other words, we have $\phi'_n(x) \leq \phi_n(x)$ for any $x \in L$. In the case when all dataflow functions are distributive—that is, $f(x \wedge y) = f(x) \wedge f(y)$—there is no loss of precision, and $\phi'_n = \phi_n$.

Phases II and III on the condensed ICFG are similar to phases II and III on the original ICFG. For any node n, the solution S'_n computed by phase III of the analysis on the condensed graph is a safe approximation of the solution S_n computed by phase III of the analysis of the original ICFG—that is, $S'_n \leq S_n$. If the dataflow functions are distributive, we have $S'_n = S_n$, and the component-level analysis achieves the same precision as the whole-program analysis.

3.3 Analysis Implementation

The approach described above provides the conceptual foundations for designing CLA versions of whole-program analyses. In order to implement an actual analysis, an analysis builder has to address two important issues.

First, the library summary should contain enough information so that the CLA of *Main* can compute a whole-program call graph, in order to construct the *inter*procedural edges in the condensed ICFG. As a simple example, for Figure 2, the summary could record the fact that the call at node 16 is through a function pointer, and that no function addresses are taken in the library. The CLA of *Main* can resolve the call at 16 to any procedure whose address is taken in *Lib* or in *Main*; for the particular main component in Figure 1, the only possible target is `ext`. As another simple example, for a Java library, the summary can store the static receiver type and static target method for each non-fixed call site. When the code of *Main* becomes available, the whole-program class hierarchy can be constructed and used to determine the potential target methods at non-fixed library calls. Of course, more sophisticated approaches for call graph construction are possible. The adaptation of these techniques to component-level analysis is beyond the scope of this paper; some existing work already solves certain instances of this problem (e.g., [24]).

A second key issue for component-level analysis is the representation, composition, and meet of dataflow functions. The function space should allow compact representation of functions. For a large number of important dataflow problems, such compact representations have already been defined. In particular, interprocedural finite distributive subset (IFDS) problems [18] and interprocedural

distributive environment (IDE) problems [27] have compact function representations and efficient functional composition and functional meet [19, 18, 27]. These two categories of problems are significant because they cover a large number of widely used interprocedural analyses [19] such as reaching definitions, available expressions, live variables, truly-live variables, possibly-uninitialized variables, several forms of constant propagation, flow-sensitive side-effects, some forms of may-alias and must-alias analysis, interprocedural slicing, 0-CFA type analysis for Java [9], field-based points-to analysis for Java [15], and object naming analysis [21]. The general theoretical approach described earlier can be instantiated to IFDS/IDE problems by using *graph-based analysis algorithms* similar to the whole-program algorithms from [19, 18, 27]. Using these techniques, it becomes possible to design CLA versions of many important and widely-used analyses.

In the particular case of the reaching definitions analysis, a function f can be represented by a pair (K, G) where K is the set of definitions killed by f, and G is the set of definitions generated by f. The functional meet of $f_1 = (K_1, G_1)$ and $f_2 = (K_2, G_2)$ is represented by $(K_1 \cap K_2, G_1 \cup G_2)$. The functional composition $f_2 \circ f_1$ corresponds to $(K_1 \cup K_2, (G_1 - K_2) \cup G_2)$.

4 Experimental Study

This section presents an experimental study which evaluates the effectiveness of the proposed approach for CLA. The study was performed on the 19 Java programs shown in Table 1. Each program was processed using the Soot frame-

Table 1. Analyzed programs

Program	User Methods	All Methods	User CFG Nodes	All CFG Nodes
`jb-6.1`	149	7130	2888	117781
`socksproxy`	113	7178	2449	118969
`jlex-1.2.6`	133	7113	7210	122095
`RabbIT2`	184	7368	3455	122755
`javacup-0.10j`	332	7312	9066	124000
`sablecc-2.18.2`	1744	8755	24149	139498
`db`	96	17755	2397	303193
`compress`	100	17760	2399	303201
`fractal`	184	17919	3526	305731
`raytrace`	219	17878	5179	305973
`socksecho`	176	17966	3562	306658
`jack`	349	18008	11541	312333
`jtar-1.21`	224	18152	6145	312562
`jess`	641	18323	12365	313375
`mpegaudio`	307	17967	14304	315094
`jflex-1.4.1`	509	18217	14826	315936
`mindterm-1.1.5`	598	18385	17792	321948
`muffin-0.9.3a`	933	18820	18383	323560
`javac`	1185	18868	25496	326524

work [30] version 2.2.2, on top of the J2SE 1.4.2 libraries. The experiments were performed on a 2.8GHz Pentium4 PC with 2GB of RAM running Sun's HotSpot Client VM version 1.4.2 using a maximum heap size of 1.5GB (JVM option `Xmx`).

For each of the data programs, we utilized Soot's call graph construction algorithm based on class hierarchy analysis. Column *User Methods* shows the number of reachable methods which are declared in the program code (i.e., all reachable non-library methods). The total number of reachable methods is listed in column *All Methods*. As Table 1 shows, the vast majority (80.1% to 99.5%) of reachable methods were contained in the libraries. These measurements clearly indicate that the cost of whole-program analysis will be dominated by the cost to analyze the relevant library code. This observation provides a very strong motivation for using summary-based component-level analysis of *Main*.

For each of the data programs, we constructed the whole-program ICFG. Columns *User CFG Nodes* and *All CFG Nodes* of Table 1 describe the number of ICFG nodes. Again, the large majority of nodes (between 82.7% and 99.2%) were in the Java libraries. Using the techniques described in Section 3.1, we constructed the "condensed" version of the ICFG. The reduction of the number of nodes (shown in Table 2) was substantial, with the condensed ICFGs containing 59.2% to 71.4% fewer nodes than the original ICFGs. Our experiments also showed that the reduction in the number of ICFG edges was equally significant; for brevity, we do not present these results. Since the cost of dataflow analyses typically depends on ICFG size, these results clearly show that a summary-based approach can achieve considerable cost reduction.

To measure the savings achieved by our technique, we implemented Sharir-Pnueli's Phase I for a variation of the reaching definitions problem for Java. Java has three types of memory locations: local variables, instance fields, and static fields. For local variables, the reaching definitions problem is purely intra-procedural. For instance fields, an alias analysis must be used to resolve indirect accesses through object references. Since such resolution is typically done with a *may-alias* analysis, field definitions cannot be killed safely; as a result, the dataflow functions are different from the ones in the classical reaching definitions

Table 2. Reduction in the number of ICFG nodes

Program	Condensed Nodes	Reduction	Program	Condensed Nodes	Reduction
`jb`	35556	69.8%	`socksecho`	88453	71.2%
`socksproxy`	35604	70.1%	`jack`	95926	69.3%
`jlex`	39876	67.3%	`jtar`	91722	70.7%
`RabbIT2`	37155	69.7%	`jess`	96817	69.1%
`javacup`	41733	66.3%	`mpegaudio`	98689	68.7%
`sablecc`	56954	59.2%	`jflex`	99376	68.6%
`db`	86783	71.4%	`mindterm`	102769	68.1%
`compress`	86789	71.4%	`muffin`	103864	67.9%
`fractal`	88195	71.2%	`javac`	109965	66.3%
`raytrace`	89563	70.7%			

Table 3. Running time of the analysis (in seconds) and % time reduction

Program	WPA	CLA	Program	WPA	CLA
jb	97.3	19.2 (80.2%)	socksecho	658.5	328.6 (50.1%)
socksproxy	96.1	20.5 (78.6%)	jack	665.2	322.6 (51.5%)
jlex	101.2	17.5 (82.8%)	jtar	682.3	349.6 (48.8%)
RabbIT2	101.1	22.1 (78.2%)	jess	665.1	334.6 (49.7%)
javacup	116.6	24.0 (79.4%)	mpegaudio	585.9	240.1 (59.0%)
sablecc	139.3	34.5 (75.2%)	jflex	686.0	454.1 (33.8%)
db	656.1	392.5 (40.2%)	mindterm	648.6	342.0 (47.3%)
compress	597.5	300.4 (49.7%)	muffin	658.1	366.1 (44.4%)
fractal	676.0	261.5 (61.3%)	javac	656.1	346.7 (47.2%)
raytrace	651.3	417.1 (36.0%)			

problem. Thus, we implemented a reaching definitions analysis for static fields only, where the dataflow functions are of the form described in Section 2.

Table 3 shows the analysis running time (in seconds) using both whole-program analysis (column *WPA*) and component-level analysis (column *CLA*). The reduction in running time ranged from 33.8% to 82.8%, with an average of 57.5%. Even though Table 3 only shows results for one particular dataflow analysis, we believe that due to the ICFG reduction, such dramatic savings will not be limited to the reaching definitions problem. Studying the effects of CLA on other dataflow analyses remains open for future investigations.

5 Related Work

Many techniques have been introduced for efficient dataflow analysis, with various representations of the flow of control and data. Examples include the elimination algorithms from [26] and the flow graph summarization of Callahan [1]. Our condensed ICFG is conceptually similar to the program summary graph from [1]. Efficient data flow representation is typically in terms of groups of problems, beginning with the slot-wise problems (e.g., [7]), eventually leading to the formulation of the IFDS and IDE frameworks [18, 27].

Various whole-program dataflow analyses construct summary information about a procedure, and then use this information when analyzing the callers of that procedure. An early example are the jump functions used for interprocedural constant propagation [10]. As another example, several analyses [2, 4, 31, 3, 25] perform a bottom-up traversal of the program call graph and compute a summary function for each visited procedure. This summary function is then used when analyzing the callers of that procedure and when constructing their summary functions. Summary functions can also be created in top-down manner, by introducing all possible contexts at the entry of the analyzed procedure [11]. Some approaches compute summary information for a software component independently of the callers and callees of that component. One particular technique is to compute partial analysis results for each component, to combine the results

for all components in the program, and then to perform the rest of the analysis work (e.g., [5, 8, 6, 12, 24]). Sometimes conservative assumptions are used instead of pre-computed summaries (e.g., [29]). Finally, there is related work on incremental and parallel dataflow analysis (e.g., [16, 14]) in which the idea of a representative problem is introduced, and in which intensive local analysis is followed by a quick postpass to recover the actual solutions.

6 Conclusions and Future Work

The use of library summaries is essential for interprocedural dataflow analysis of modern software systems that are built with large library components. We propose a general theoretical framework for summary-based analysis, and present initial results that strongly indicate the potential of this technique to reduce analysis cost. In future work, we will (1) instantiate the framework to a range of popular dataflow analyses, starting with IFDS and IDE analyses, and (2) implement and evaluate these analyses, in order to gather experimental evidence of the benefits of the proposed approach. We will also consider systems built with multiple library components (e.g., libraries that use other libraries).

Acknowledgment. We would like to thank the CC reviewers for their helpful comments and suggestions.

References

1. D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Conf. Programming Language Design and Implementation*, pages 47–56, 1988.
2. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symp. Principles of Programming Languages*, pages 133–146, 1999.
3. B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conf. Programming Language Design and Implementation*, 2000.
4. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.
5. M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Symp. Principles of Programming Languages*, pages 451–464, 1993.
6. M. Das. Unification-based pointer analysis with directional assignments. In *Conf. Programming Language Design and Implementation*, pages 35–46, 2000.
7. D. Dhamdhere, B. Rosen, and K. Zadeck. How to analyze large programs efficiently and informatively. In *Conf. Programming Language Design and Implementation*, pages 212–223, 1992.
8. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Programming Languages and Systems*, 21(2):370–416, Mar. 1999.
9. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Programming Languages and Systems*, 23(6):685–746, Nov. 2001.
10. D. Grove and L. Torczon. Interprocedural constant propagation: a study of jump function implementation. In *Conf. Programming Language Design and Implementation*, pages 90–99, 1993.

11. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Tran. Software Engineering*, 22(7):442–460, July 1996.
12. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA. In *Conf. Programming Language Design and Implementation*, pages 254–263, 2001.
13. F. C. Kuck. Class analysis for extensible Java software. Master's thesis, Ohio State University, Sept. 2004.
14. Y.-F. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In *Int. Conf. Supercomputing*, pages 236–247, 1992.
15. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Int. Conf. Compiler Construction*, LNCS 2622, pages 153–169, 2003.
16. T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Symp. Principles of Programming Languages*, pages 184–196, 1990.
17. A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for C programs with function pointers. *Int. J. Automated Software Engineering*, 11(1):7–26, 2004.
18. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. Principles of Programming Languages*, p. 49–61, 1995.
19. T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, Apr. 1994.
20. A. Rountev. Component-level dataflow analysis. In *Int. Symp. Component-Based Software Engineering*, LNCS 3489, pages 82–89, 2005.
21. A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *Int. Conf. Software Engineering*, pages 254–263, 2005.
22. A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. Technical Report OSU-CISRC-1/06-TR01, Jan. 2006.
23. A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Tran. Software Engineering*, 30(6):372–387, June 2004.
24. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Int. Conf. Compiler Construction*, LNCS 2027, pages 20–36, 2001.
25. E. Ruf. Effective synchronization removal for Java. In *Conf. Programming Language Design and Implementation*, pages 208–218, 2000.
26. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, 1986.
27. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Comp. Sci.*, 167:131–170, 1996.
28. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. 1981.
29. F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for Java. *ACM Trans. Programming Languages and Systems*, 24(6):625–666, 2002.
30. R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Int. Conf. Compiler Construction*, LNCS 1781, pages 18–34, 2000.
31. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.