An Improved Bound for Call Strings-Based Interprocedural Analysis of Bit Vector Frameworks

BAGESHRI KARKARE and UDAY KHEDKER
IIT Bombay

Interprocedural data flow analysis extends the scope of analysis across procedure boundaries in search of increased optimization opportunities. Call strings based approach is a general approach for performing flow and context sensitive interprocedural analysis. It maintains a history of calls along with the data flow information in the form of call strings, which are sequences of unfinished calls. Recursive programs may need infinite call strings for interprocedural data flow analysis. For bit vector frameworks this method is believed to require all call strings of lengths up to $3K$, where K is the maximum number of distinct call sites in any call chain.

We combine the nature of information flows in bit-vector data flow analysis with the structure of interprocedurally valid paths to bound the call strings. Instead of bounding the length of call strings, we bound the number of occurrences of any call site in a call string. We show that the call strings in which a call site appears at most three times, are sufficient for convergence on interprocedural maximum fixed point solution. Though this results in the same worst case length of call strings, it does not require constructing all call strings up to length $3K$. Our empirical measurements on recursive programs show that our bound reduces the lengths and the number of call strings, and hence the analysis time, significantly.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*; F.2 [**Analysis of Algorithms and Problem Complexity**]

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Interprocedural Data Flow Analysis, Bit Vector Data Flow Frameworks

ACM Reference Format:

Karkare, B. and Khedker, U. 2007. An improved bound for call strings based interprocedural analysis of bit vector frameworks ACM Trans. Program. Lang. Syst. 29, 6, Article 38 (October 2007), 13 pages. DOI = 10.1145/1286821.1286829 http://doi.acm.org/10.1145/1286821.1286829

B. Karkare was supported by Infosys Technologies Limited, Bangalore, under Infosys Felowship Award.

Author's address: U. Khedker: uday@cse.iitb.ac.in

1. INTRODUCTION

Interprocedural data flow analysis extends the scope of analysis across procedure boundaries in search of increased optimization opportunities. It is desirable that such an analysis should be sensitive to the contexts in which procedures execute. Let c_f denote a particular call to a procedure f. Then the context of c_f is captured by the call stack. A typical static representation of program used for interprocedural analysis is a *supergraph* that connects individual program flow graphs with call and return edges. The supergraph admits infeasible contexts arising out of control flow paths that violate nestings of matching call-return pairs. The precision of interprocedural analysis requires excluding infeasible contexts arising out of interprocedurally invalid paths and distinguishing between distinct contexts arising out of distinct interprocedurally valid paths. Further, in the presence of recursion, there may be infinite contexts and feasibility of analysis requires a static summarization of contexts.

The generic interprocedural data flow analysis methods can be divided into two broad categories. The *functional approaches* compute effective flow functions for procedures in a *context-independent* manner and then these flow functions are plugged into different contexts for performing analysis. The degree of context sensitivity of the resulting analyses depends on the precision of the effective flow functions computed. Functional approach [Sharir and Pnueli 1981], graph reachability approach [Reps et al. 1995], interprocedural constant propagation [Callahan et al. 1986], etc. use this strategy. The *value-based approaches* maintain data flow values for different contexts. They may have to create special representations of the program for this purpose. The degree of context sensitivity of resulting analyses depends on the extent of approximation of contexts. The call strings approach [Sharir and Pnueli 1981], Precise interprocedural analysis algorithm [Myers 1981], Program analyzer generator (PAG) [Alt et al. 1995; Alt and Martin 1995], Interprocedural Reaching Definitions in presence of single level pointers [Pande et al. 1992], cloning based interprocedural pointer analysis [Whaley and Lam 2004], etc. use this strategy.

The call string approach to interprocedural data flow analysis [Sharir and Pnueli 1981] embeds contexts in the data flow information in form of call strings. For nonrecursive programs, all call strings of length up to K are required where K is the number of distinct call sites in a call chain. For recursive programs, termination of this method requires all call strings of length up to $K \cdot (|L| + 1)^2$ where L denotes the lattice of data flow values. This bound reduces to $K \cdot (|\widehat{L}| + 1)^2$ for separable frameworks where $\widehat{L}$ is the component lattice. For bit vector frameworks, although $|\widehat{L}| = 2$, this bound further reduces to $3K$ [Sharir and Pnueli 1981].

We restrict the focus of this article to the call strings approach for bit-vector frameworks and propose a different way of bounding call strings. We study the influence of interprocedural validity constraints on information flows and bound the number of occurrences of any call site in a call string instead of bounding the length. In particular we show that only those call strings in which any call site occurs at most three times are sufficient for computing precise solution of bit-vector frameworks. This is a program independent bound in

contrast to the length bound of $3K$ which involves either counting the number of distinct call sites in all call chains to determine K, or using the largest value of K which is the total number of call sites in a program. Although the worst case length of call strings in our bound is the same as the traditional length bound, not all call strings of lengths up to $3K$ need to be constructed. Our empirical measurements on recursive programs show that this bound reduces the lengths and number of call strings and hence the analysis time significantly.

Interestingly this does not change the call string-based interprocedural data flow analysis algorithm except that the criterion of deciding whether a longer call string should be constructed by appending a call site c_i to a call string σ is now based on the number of occurrences of c_i in σ rather than the length of σ.

The rest of the article is organized as follows. Section 2 reviews the background, while sections 3 and 4 explain the new bound. Section 5 provides empirical measurements and Section 6 concludes the paper.

2. BACKGROUND

This section reviews some basic concepts of data flow analysis and the call strings–based interprocedural data flow analysis.

2.1 Bit-Vector Data Flow Frameworks

Bit vector frameworks [Aho et al. 1986; Graham and Wegman 1975; Hecht 1977; Kam and Ullman 1977; Khedker 2002; Kildall 1973; Muchnick 1997] (called *1-related* by Sharir and Pnueli [1981]) are data flow frameworks characterized by following properties:

(1) Lattice: The overall lattice L_D is a product $\widehat{L}_1 \times \widehat{L}_2 \times \cdots \times \widehat{L}_\delta$ of component lattices $\widehat{L}_i = \{1, 0\}$ that contain the data flow values of individual entities. The *top* ($\widehat{\top}$) and *bottom* ($\widehat{\bot}$) elements are 1 or 0 depending upon the confluence operation. The *height* of $\widehat{L}_i$ is 1. the overall lattice is δ.

(2) Flow functions: The flow functions $h : L_D \to L_D$ can be written as

$$h(X) = \mathit{Gen} + (X - \mathit{Kill}) \quad \text{where} X, \mathit{Gen}, \mathit{Kill} \in L_D \text{ and } \mathit{Gen} \text{ and } \mathit{Kill} \text{ are constant .}$$

As a consequence that is, the flow functions are *separable* (called *decomposable* by Sharir and Pnueli [1981]); that is, they are tuples of *bit functions* $\widehat{h} : \widehat{L} \to \widehat{L}$ which operate individually on the component lattices.

$\widehat{h}(\widehat{X}) = \widehat{\mathit{Gen}} + (\widehat{X} - \widehat{\mathit{Kill}})$ where $\widehat{X}, \widehat{\mathit{Gen}}, \widehat{\mathit{Kill}} \in \widehat{L} = \{1, 0\}$ and $\widehat{\mathit{Gen}}$ and $\widehat{\mathit{Kill}}$ are constant.

As shown below, these functions are either constant functions or identity functions. Besides, $\widehat{h}(\widehat{h}(\widehat{X})) = \widehat{h}(\widehat{X})$ in each case.

Gen	*Kill*	$\widehat{h}(\widehat{X})$	Name of $\widehat{h}$	Type of $\widehat{h}$
1	1	1	*Raise*	Constant function
1	0	1	*Raise*	Constant function
0	1	0	*Lower*	Constant function
0	0	$\widehat{X}$	*Propagate*	Identity function

Available expressions analysis, reaching definitions analysis, live variables analysis, etc. are classical examples of bit-vector frameworks [Aho et al. 1986; Hecht 1977].

2.2 Computing Maximum Fixed Point Solution for Bit Vector Frameworks

Since bit vector frameworks have only constant and identity functions and $\widehat{L}$ is $\{1, 0\}$, computing Maximal Fixed Point solution at intraprocedural level involves the following steps [Khedker and Dhamdhere 1994; Khedker 2002]:

(1) *Initializing data flow values at all program points to* $\top$. The program entry or exit may be initialized to some other values.
(2) *Computing data flow information by applying* ***Raise*** / ***Lower*** *functions*. Since ***Raise*** and ***Lower*** are constant functions, the first application of ***Raise***/***Lower*** function at a program point i computes the final value of corresponding entity at i. We call this *information generation*. Subsequent applications of the flow function compute the same value. Note that multiple occurrences of ***Raise***/***Lower*** functions along a path imply generating information at distinct program points independently.
(3) *Propagating information to all possible program points*. Information generated at a program point must be propagated to other program points through a series of identity functions. After information generation at all possible program points is complete, *information propagation* ensures computation of final information at all program points. At an abstract level the propagation can be seen as a problem of graph reachability, although the graph reachability algorithms [Reps et al. 1995] for interprocedural analysis do not have much resemblance with propagation.

These concepts can be used to explain the classical complexity result of at most $1 + d$ iterations required for intraprocedural data flow analysis of unidirectional bit vector frameworks [Aho et al. 1986; Hecht 1977; Khedker 2002; Muchnick 1997] as follows:

> The first iteration is sufficient to generate data flow information at all possible program points in the control flow graph. Since there are at most d back edges in any acyclic path, at most d additional iterations are required to propagate the information to other program points.

2.3 Call Strings–Based Interprocedural Data Flow Analysis

A *call string* at a program point p is a sequence $c_1c_2 \ldots c_k$ of unfinished calls when execution reaches p. λ denotes an empty call string. Call strings at p can be used to distinguish between information propagated along different interprocedural paths reaching p. Merging the data flow values propagated along all call strings reaching each program point results in a meet-over-all-interprocedurally-valid-paths solution.

For interprocedural analysis, a program is represented by a *supergraph* by connecting the control flow graphs of the individual procedures. Let ***Entry*** and ***Exit*** denote the entry and exit of the main program. They form the entry and

exit nodes of the supergraph. Let $Start_p$ and End_p denote the start and end of procedure p. A call site C_i is split into a call node c_i and the corresponding return node r_i. If C_i calls procedure p, then the *interprocedural* edges $c_i \rightarrow Start_p$ and $End_p \rightarrow r_i$ are added.

Data flow analysis of a program involves traversing its supergraph. When intraprocedural edges are traversed, the data flow values are modified by the flow functions. When a call edge $c_i \rightarrow Start_p$ is traversed, the data flow values are propagated to $Start_p$. Propagating the data flow values along return edge $End_p \rightarrow r_i$ implies propagating the information back to the caller from which the callee was reached during analysis. Such controlled manipulation and propagation is achieved by creating a new data flow value $\langle \sigma,\ d \rangle$ where σ is a call string reaching the program point under consideration and d is the original data flow value reaching the point along σ. When $\langle \sigma,\ d \rangle$ is propagated further, a call/return edge modifies σ only, whereas an intraprocedural edge may modify d only.

Call-string construction is governed by the interprocedural edges in the supergraph. Let σ be a call string reaching node m in some procedure p. For an intraprocedural edge $m \rightarrow n$, σ reaches n unmodified. For a call edge $m \rightarrow n$ where m is c_i and n is $Start_q$, call string $\sigma \cdot c_i$ reaches $Start_q$. For a return edge $m \rightarrow n$ where m is End_p and n is r_i, if $\sigma \equiv \sigma' \cdot c_i$, then σ' reaches r_i.

3. IDENTIfiCATION OF RELEVANT PATHS

An interprocedurally valid path may contain multiple subpaths starting and ending in the same procedure.[1] Such subpaths start at different call sites in the procedure and have different call strings associated with them. We use a tree representation to distinguish between these subpaths to identify the call strings sufficient for data flow analysis.

3.1 Path-Tree

A *path-tree* of an interprocedurally valid path ρ is defined as follows: Each node in the path-tree corresponds to a call to a procedure and is defined as a 3-tuple $\langle$*prefix, subtree, suffix*$\rangle$ where *prefix* is the call site that calls the procedure, *subtree* is a sequence of child nodes representing different subpaths starting at different call sites in that procedure, and *suffix* is the return node corresponding to *prefix*. A leaf node indicates call to a leaf level procedure and hence contains a *NULL* subtree.

The supergraph path $(Entry, n_1, c_1, s_p, n_3, c_2, s_q, n_5, e_q, r_2, n_4, e_p, r_1, n_2, Exit)$ in Figure 1 can be abstracted as (c_1, c_2, r_2, r_1) and its path-tree contains a root node $\langle c_1, SUBTREE, r_1 \rangle$ with *SUBTREE* being a single leaf node $\langle c_2, NULL, r_2 \rangle$. Consider a more complex abstract path from Figure 1: $(c_1, c_2, c_4, c_2, c_3, c_2, r_2, r_3, c_5, r_5, r_2, r_4, r_2, r_1)$. The tree representation of this path is shown in Figure 2.

A call string starts at the root node of a path-tree and concatenation of node prefixes generates longer call strings. A call string corresponding to a leaf node

[1]A similar concept is presented as Same-level-valid-paths in Reps et al. [1995].

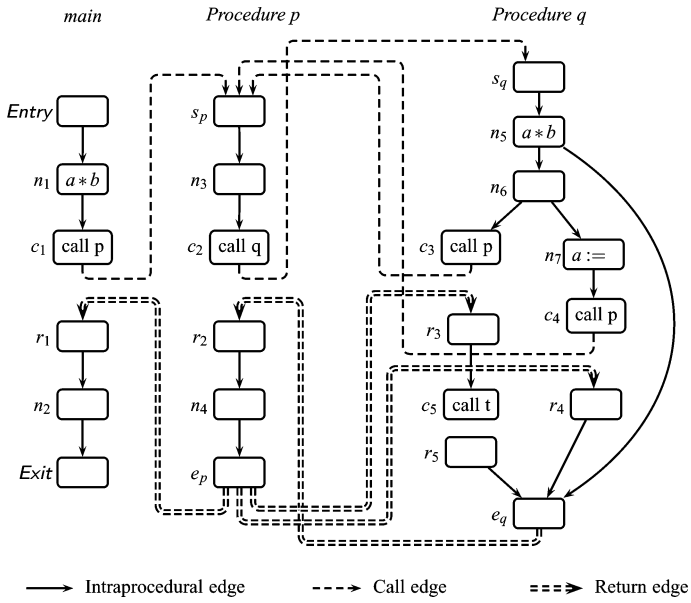

Fig. 1. A supergraph showing only the relevant program statements. Procedure t is a leaf level procedure.

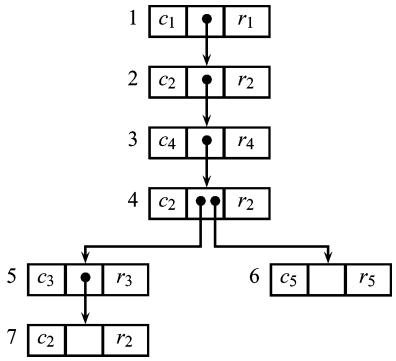

Fig. 2. A path-tree for a call return sequence $c_1, c_2, c_4, c_2, c_3, c_2, r_2, r_3, c_5, r_5, r_2, r_4, r_2, r_1$.

is a maximal call string. A call string reaching a nonleaf node is a common prefix of all call strings corresponding to the successors of that node. During data flow analysis, the data flow information at the end of a subpath corresponding to a successor of a path-tree node is propagated to the following subpath corresponding to the next successor by this common call string. For example, in Figure 2(a), call string $c_1c_2c_4c_2$ holds the data flow information before and after performing data flow analysis along the subpath starting at c_3 and ending at r_3 represented by nodes 5 and 7. This information is propagated to the subpath starting at c_5 (represented by node 6 in the path tree). Further the same call string is used for recording the data flow information at r_5.

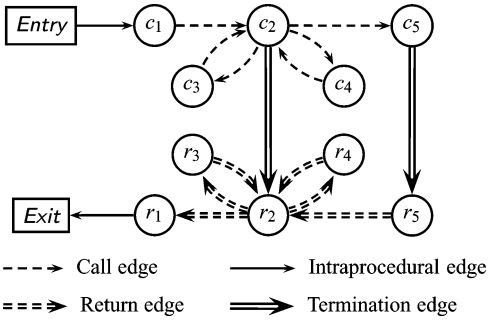

Fig. 3. The call-strings graph for supergraph in Figure 1.

3.2 Call-Strings Graph

In order to bound the length of call strings for data flow analysis, we consider information flow along all valid call-return sequences in all possible path-trees such that these sequences correspond to distinct call strings. We represent these sequences in the form of a graph called *call-strings graph* (CSG), which is then used to show the proposed bound.

Nodes of a CSG include the *Entry*, *Exit*, and all call and return nodes (i.e. all c_i and r_i). An edge from call node c_i to call node c_j represents the fact that c_j is a call site in the function called by call site c_i. For every *call edge* $c_i \rightarrow c_j$, there is a corresponding *return edge* $r_j \rightarrow r_i$. A *termination edge* $c_i \rightarrow r_i$ is present when there exists a call-free path in the function called by call site c_i. Figure 3 shows the CSG for the supergraph in Figure 1. Call strings are defined by CSG-paths consisting of call nodes only. Since CSG paths represent valid call-return sequences in path-trees such that these sequences correspond to distinct call strings, it does not have a path from a return node to any call node.

Note that the Call-Strings Graph is neither required for performing data flow analysis nor can be used for that purpose. It is a construction used for proving our bound on call strings. Since it covers all relevant interprocedural paths, it is sufficient to restrict the analysis of information flows to CSG in order to identify the relevant call strings. We use CSG paths to bound the set of call strings and show that call strings with at most three occurrences of any call site are sufficient to cover all information flows.

4. A PROGRAM INDEPENDENT BOUND ON CALL STRINGS

This section shows that a maximal fixed point solution of interprocedural analysis of bit-vector problems can be computed using call strings with at most three occurrences of any call site.

If information flow can be restricted to interprocedurally valid paths, the generation-propagation approach of MFP computation for bit-vector frameworks discussed in Section 2.2 can be directly extended from intraprocedural to interprocedural level. In bit-vector frameworks, information generation takes place during the first visit to each node in the supergraph and the validity constraints are irrelevant. However, information propagation must exclude propagation along interprocedurally infeasible paths. For the purpose

of bounding the call strings, we conservatively assume all possible information flows by considering all possible combinations of source and targets of data flow information. Given m nodes in the supergraph, we consider m^2 information flows from

(1) each node as a potential information source, to
(2) each node as a potential information target.

When a node is considered as source, for maximum information flow, we assume that only that node has a nonidentity flow functions and the flow functions of all other nodes are identity functions.

For exhaustive information propagation (i.e., computing the maximum fixed point solution) at interprocedural level, we must ensure that all interprocedurally feasible paths propagating information between all pairs of source and target nodes are covered. Also, in the case of cyclic paths, it is desirable to avoid unnecessary repetitions of cycles. We use these criteria for discovering relevant call strings.

4.1 Bounding Call Strings using CSG

For recursive programs, CSG has infinite number of paths. We restrict the set of call strings by selecting the CSG paths that are interprocedurally valid, and which exclude unnecessary repetitions of cycles. The cycles which do not influence information propagation are eliminated on the basis of the following obvious properties:

—For every cyclic path from node n to m, there exists an acyclic path from n to m.
—If all flow functions along a cyclic path from node n to m are identity functions then there exists an acyclic path from n to m along which all flow functions are identity functions.

We denote the set of all CSG paths from to node n to node m by $P(n \rightsquigarrow m)$. If these paths are required to pass through some node l, they are denoted by $P(n \rightsquigarrow l \rightsquigarrow m)$. As a logical extension of the notation, if they are required to pass through node o after passing through node l, they are denoted by $P(n \rightsquigarrow l \rightsquigarrow o \rightsquigarrow m)$. The set of all acyclic paths from node n to node m are denoted by $P_A(n \rightsquigarrow m) \subseteq P(n \rightsquigarrow m)$.

The information flow consisting of the generation of data flow information at source S followed by its propagation to target T takes place along paths $\rho \in P(\mathit{Entry} \rightsquigarrow S \rightsquigarrow T)$. For a given source target pair S and T, we assume that S is the only node with nonidentity flow function and all other flow functions along ρ are identity functions. If there is some node S' along ρ that has a constant transfer function, then there are two cases:

—If S' appears to the left of S on ρ, then the presence of S' does not influence the information flow from S to T.
—If S' appears between S and T on ρ, then there is no information flow from S to T along ρ and considering ρ for information flow is a conservative

assumption for the purpose of bounding call strings. Also, the paths in $P(\mathit{Entry} \rightsquigarrow S' \rightsquigarrow T)$ are considered separately for the information flow from S' to T.

The call strings required for exhaustive information propagation can be determined by analyzing the information flows between all possible source-target pairs along valid CSG paths. Following cases exhaustively cover all possible source-target pairs:

(1) *Source is a call node c_S and target is also some call node c_T.* In this case, the set of paths is $P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow c_T)$.[2] Since there are no return nodes in this pattern, the issue of interprocedural validity in terms of matching calls and returns does not arise. Information generated at c_S can reach c_T via acyclic paths (recall that cyclic paths need not be traversed for information propagation). Also, since the paths from *Entry* to c_S are required only for reaching c_S and not for information propagation, it is sufficient to consider acyclic paths from *Entry* to c_S. Thus call strings required for information flow along $\rho \in P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow c_T)$, can be constructed by concatenating the paths in $P_A(\mathit{Entry} \rightsquigarrow c_S)$ with the paths in $P_A(c_S \rightsquigarrow c_T)$. These concatenated acyclic segments may have some common node(s).[3] However, the common nodes do not occur more than twice in the entire call string. Thus call strings with at most two occurrences of any call site are sufficient to propagate information from one call node to another call node.
For example, consider the CSG in Figure 3. In order to propagate information generated at c_4 to c_3, a call string $c_1c_2c_4c_2c_3$ is required. This call string contains two occurrences of c_2.

(2) *Source is a call node c_S and target is some return node r_T.* In this case, the CSG paths are contained in $P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow c_T \rightsquigarrow r_T) \cup P(\mathit{Entry} \rightsquigarrow c_T \rightsquigarrow c_S \rightsquigarrow r_S \rightsquigarrow r_T)$. Note that c_T and r_S appear due to validity constraints. The call strings for the two sets are constructed as follows:
   (a) $P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow c_T \rightsquigarrow r_T)$. In this case, the call strings are derived from the paths in $P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow c_T \rightsquigarrow c_L)$ where c_L is the last call node in the paths. Thus each path can be divided into three segments contained in $P_A(\mathit{Entry} \rightsquigarrow c_S)$, $P_A(c_S \rightsquigarrow c_T)$, and $P_A(c_T \rightsquigarrow c_L)$ respectively. The relevant call strings can be constructed by concatenating acyclic paths along these segments. Since a call node may be common to all three segments, a call node can occur at most three times in a call string corresponding to this case.
   (b) $P(\mathit{Entry} \rightsquigarrow c_T \rightsquigarrow c_S \rightsquigarrow r_S \rightsquigarrow r_T)$. In this case, the call strings are derived from the paths in $P(\mathit{Entry} \rightsquigarrow c_T \rightsquigarrow c_S \rightsquigarrow c_L)$, where c_L is the last call node in the path. Similar to case (a) above, there are three acyclic segments and a call node can occur at most three times in a call string corresponding to this case also.

[2]Note that c_S and c_T need not be distinct.
[3]A call node belongs to that segment in which it appears anywhere after the first node, i.e. S belongs all paths in $P_A(\mathit{Entry} \rightsquigarrow S)$. It does not belong to any path in $P_A(S \rightsquigarrow T)$.

For example, for the CSG in Figure 3, information propagation from c_4 to r_3 requires following paths:

- $c_1 \rightarrow c_2 \rightarrow c_4 \rightarrow c_2 \rightarrow c_3 \rightarrow c_2 \rightarrow r_2 \rightarrow r_3$: This path corresponds to category (a). It requires call string $c_1c_2c_4c_2c_3c_2$ containing three occurrences of c_2; and
- $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_2 \rightarrow c_4 \rightarrow c_2 \rightarrow r_2 \rightarrow r_4 \rightarrow r_2 \rightarrow r_3$: This path corresponds to category (b) and requires call string $c_1c_2c_3c_2c_4c_2$ which requires three occurrences of c_2. □

When source and target are call and return points corresponding to the same call site, the relevant paths are contained in $P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow r_S)$. The call strings are derived from the paths in $P(\mathit{Entry} \rightsquigarrow c_S \rightsquigarrow c_L)$. These call strings contain two concatenated acyclic segments as in case (1) above, resulting in at most two occurrences of any call site.

(3) *Source is a return node r_S and target is also some return node r_T.* The relevant paths are contained in $P(\mathit{Entry} \rightsquigarrow c_T \rightsquigarrow c_S \rightsquigarrow r_S \rightsquigarrow r_T)$. This is same as case 2(b).

Thus we conclude that for the purpose of interprocedural analysis of bit-vector frameworks, it is sufficient to construct call strings in which no call site occurs more than three times. Since the information flow along these call strings follows the generation-propagation approach, convergence on MFP solution is guaranteed.

5. EMPIRICAL MEASUREMENTS

We have performed interprocedural Reaching Definitions analysis on following recursive programs: `Hanoi`,[4] `queens`,[5] `mergesort`,[6] `bit_gray`,[7] `181.mcf` from SPEC-2000, `fourinarow`,[8] and the test example taken from Sharir and Pnueli [1981], which is same as the example in Figure 1 without the call site c_5. We have incorporated the interprocedural analysis into gcc 4.0 as an additional pass that constructs supergraph and performs the call strings based analysis on the Gimple IR. These experiments were carried out on a P4 (3.06GHz) machine with 1GB RAM running Fedora Core 6.

The analysis was performed using the $3K$ length bound as well as our 3 occurrences bound. Table I shows the comparison of lengths and number of call strings constructed using both approaches, and the analysis time. Note that although we have reported the code size in lines of code, the number of call sites is a more relevant measure for size of the program for the purpose of these experiments since number of call strings depend upon the number of call sites, rather than the overall program size. For the purpose of experimentation we had to restrict the number of call strings to 2×10^5 for $3K$ bound. This was done primarily due to the compiler running out of space. As is clear from the

[4]From `http://www.ece.cmu.edu/~ece548/hw/lab1/hanoi.c`
[5]From `http://home.iae.nl/users/mhx/queens.c`
[6]From `http://linux.wku.edu/~lamonml/algor/sort/merge.html`
[7]From `http://paul.rutgers.edu/~rhoads/Code/bit_gray.c`
[8]From FreeBench v1.03 suite

Table I. Empirical Measurements

Program	LoC	#CS	$3K$ length bound				Max. 3 occurrences bound		
			K	#CSTR	Len	Time (ms)	#CSTR	Len	Time (ms)
`test`	25	4	3	47	9	1.93	23	7	1.56
`hanoi`	33	4	4	200000+	12	3973×10^3	2179	10	171.08
`bit_gray`	53	11	7	200000+	21	2705×10^3	3770	14	357.44
`merge`	73	5	5	24576	15	1121×10^3	75	8	400.7
`queens`	74	4	3	26	9	3.50	13	5	1.19
`fourinarow`	676	45	5	510	15	397.76	282	9	123.714
`181.mcf`	1299	24	6	32789	18	484×10^3	91	10	7.67

LoC denotes lines of code, #CS the number of call sites, #CSTR the total number of call strings (a + indicates that the actual number is larger than the specified number) and Len the maximum length of a call string. The analysis time is in milliseconds.

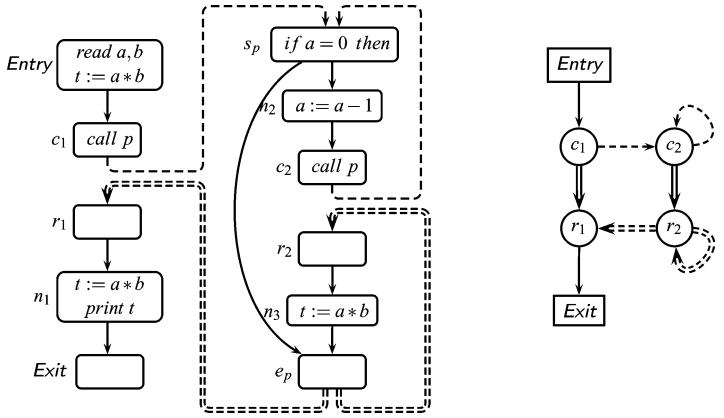

Fig. 4. A supergraph and its CSG. Though $K = 2$, call strings of length 6 are not required.

empirical measurements, our bound results in much smaller and much fewer call strings and much reduced analysis time for practical programs.

6. CONCLUSIONS AND FUTURE WORK

Poor scalability makes the call strings–based approach to interprocedural data flow analysis impractical. We have shown that instead of bounding the call string length to $3K$, bounding the number of occurrences of call sites to 3 is sufficient for the purpose of data flow analysis. This avoids computation of K and results in much smaller and much fewer call strings as corroborated by our empirical measurements over practical programs containing recursion. Observe that this does not change the call strings based interprocedural data flow analysis algorithm except that the criterion of deciding whether longer call strings should be constructed is now based on the number of occurrences of call site in a call string rather than its length.

On the theoretical side, this paper uncovers some key insights about the nature of call strings required for interprocedural bit-vector data flow analysis. As a consequence, we can explains some observations which could not be explained before. Sharir and Pnueli [1981] state that for the program in Figure 4, $K = 2$

and call strings of length up to 6 should be constructed. However, they observe that call strings longer than length 2 are not required but fail to explain the reason. We explain it as follows: The call strings graph for this program does not have any interprocedurally valid path that contains three acyclic segments all having a common node. For such programs, the bound on maximum number of occurrences of any call site in a call string can be reduced to 2 or 1 by determining the maximum number of times a call node actually appears in the acyclic segments of CSG-paths. Such a bound is a program dependent bound. In future we propose to explore this idea in detail. We would also like to extend the proposed ideas to non-bit-vector frameworks like points-to analysis [Emami et al. 1994; Kanade et al. 2005].

ACKNOWLEDGMENTS

We would like to thank Amitabha Sanyal and Supratim Biswas for many intense discussions driven by their pointed questions. Implementation of the interprocedural analysis was carried out by Divya Krishnan and Seema Ravandale. We also thank the referees for their critical reviews, which helped in improving this paper.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Alt, M. and Martin, F. 1995. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of Static Analysis Symposium (SAS'95)*, Lecture Notes in Computer Science, vol. 983, Springer, 33–50.

Alt, M., Martin, F., and Wilhelm, R. 1995. Generating Dataflow Analyzers with PAG. Tech. rep. A 10/95, Universität des Saarlandes.

Callahan, D., Cooper, K. D., Kennedy, K., and Torczon, L. 1986. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Contruction (SIGPLA)*. 152–161.

Emami, M., Ghiya, R., and Hendren, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*. 242–256.

Graham, S. L. and Wegman, M. 1975. A fast and usually linear algorithm for global flow analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 22–34.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs*. Elsevier Science Inc.

Kam, J. B. and Ullman, J. D. 1977. Monotone data flow analysis frameworks. *Acta Informatica. 7,* 3, 305–318.

Kanade, A., Khedker, U. P., and Sanyal, A. 2005. Heterogeneous fixed points with application to points-to analysis. In *APLAS*. 298–314.

Khedker, U. P. 2002. Data flow analysis. In *The Compiler Design Handbook*. CRC Press, 1–59.

Khedker, U. P. and Dhamdhere, D. M. 1994. A generalized theory of bit vector data flow analysis. *ACM Trans. Program. Lang. Syst. 16,* 5, 1472–1511.

Kildall, G. A. 1973. A unified approach to global program optimization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 194–206.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Myers, E. M. 1981. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 219–230.

PANDE, H., LANDI, W., AND RYDER, B. 1992. Interprocedural reaching definitions in the presence of single level pointers. Tech. rep. lost-tr-193, Laboratory for Computer Science Research Rutgers University.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 49–61.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall Inc.

WHALEY, J. AND LAM, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press.

Received November 2005; revised October 2006; accepted March 2007