Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis:
Resurrecting the Classical Call Strings Method

Uday P. Khedker and Bageshri Karkare

Indian Institute of Technology, Bombay

Abstract. The full call strings method is the most general, simplest, and most precise method of performing context sensitive interprocedural data flow analysis. It remembers contexts using call strings. For full precision, all call strings up to a prescribed length must be constructed. Two limitations of this method are (a) it cannot be used for frameworks with infinite lattices, and (b) the prescribed length is quadratic in the size of the lattice resulting in an impractically large number of call strings. These limitations have resulted in a proliferation of ad hoc methods which compromise on generality, precision, or simplicity.
We propose a variant of the classical full call strings method which reduces the number of call strings, and hence the analysis time, by orders of magnitude as corroborated by our empirical measurements. It reduces the worst case call string length from quadratic in the size of the lattice to linear. Further, unlike the classical method, this worst case length need not be reached. Our approach retains the precision, generality, and simplicity of call strings method without imposing any additional constraints. It can accommodate demand-driven approximations and hence can be used for frameworks with infinite lattices.

1 Introduction

Interprocedural data flow analysis extends the scope of analysis across procedure boundaries. A *context insensitive* interprocedural analysis does not distinguish between different calling contexts of a procedure and merges the data flow information across all contexts. *Context sensitive* analysis maintains separate data flow information for distinct contexts for each procedure call and hence typically computes a more precise solution.

The full call strings method [22] is the most general, simplest, and most precise method of performing context sensitive interprocedural data flow analysis. It represents context information in the form of a call string. For full precision, all call strings up to a prescribed length have to be constructed. Two limitations of this method are (a) it cannot be used for frameworks with infinite lattices, and (b) the prescribed length is quadratic in the size of the lattice resulting in an impractically large number of call strings. These limitations have resulted in a proliferation of ad hoc methods which compromise on generality, precision, or simplicity.

We modify the full call string method by identifying contexts which need not be explicitly maintained. This reduces the number of contexts dramatically without compromising on the precision, generality, and simplicity of the method. The worst case

call string length is reduced from quadratic in the size of the lattice to linear. Further, unlike the original method, our variant does not need to construct all call strings up to the worst case length. Since it can accommodate demand-driven approximations, it can be used for frameworks with infinite lattices also. Our empirical measurements show a dramatic reduction in the number of call strings and analysis time.

Interestingly, we achieve all of the above by a very simple change in the original method without affecting the essential principles of the method: In our variant, the termination of call string construction is based on the equivalence of data flow values instead of prescribed lengths. This allows us to discard call strings where they are redundant, and regenerate them when required. For cyclic call strings, regeneration facilitates iterative computation of data flow values without explicitly constructing most of the call strings. This is based on interesting insights which are explained intuitively and proved formally in this paper.

The rest of the paper is organized as follows: Section 2 provides the background, Section 3 investigates the reasons of inefficiency in call strings method and sets the stage for Section 4 which proposes our variant. Section 5 compares our work with other approaches to interprocedural analysis. Section 6 presents the empirical data and Section 7 concludes the paper.

2 Background

This section discusses safety, precision, and efficiency in the interprocedural analysis and reviews the original call strings method.

Safety, Precision, and Efficiency in Data Flow Analysis. Data flow analysis examines static representations of programs. Some paths in these representations may not correspond to valid execution paths. Some path may be valid execution paths but may be irrelevant because their analysis may not result in new information. Safety of analysis can be ensured by covering all valid paths; excluding a valid path may result in an unsafe solution. Precision can be ensured by restricting the analysis only to valid paths; including invalid paths may result in an imprecise solution. Efficiency can be ensured by restricting the analysis to relevant paths.

A *flow sensitive* intraprocedural analysis honours the control flow and computes possibly different data flow information for each program point. A *flow insensitive* analysis does not consider the control flow and hence computes imprecise (but safe) solution. A flow sensitive method excludes spurious paths and hence computes more precise solutions. A flow insensitive analysis merely accumulates the information and hence requires a single pass over a control flow graph.

Interprocedural data flow analysis is usually performed on a *supergraph* which connects control flow graphs of different procedures with call and return edges. It contains control flow paths which violate nestings of matching call return pairs. An interprocedurally valid path is a feasible execution path containing a legal sequence of call and return edges. A *context sensitive* interprocedural analysis retains distinct calling contexts to ensure propagation of information from the callee to appropriate callers. This involves restricting the analysis to interprocedurally valid paths. A *context insensitive*

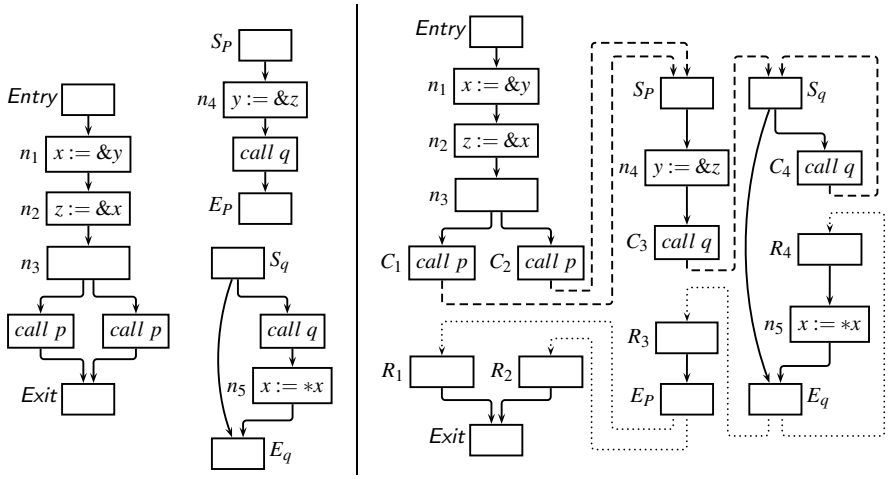

Fig. 1. Control flow graphs of recursive procedures and the corresponding supergraph.

analysis does not distinguish between valid and invalid paths and computes safe, but imprecise solution compared to a context sensitive analysis. For maximum statically achievable precision, context sensitive analysis must also be flow sensitive at the intraprocedural level. Efficiency of context sensitive interprocedural analysis requires restricting the number of contexts without merging information across distinct contexts. Context insensitive analysis effectively restricts the number of contexts to one and thus is much more efficient than context sensitive analysis.

The Call Strings Approach. The full call strings method embeds context information in the data flow information. It treats procedure calls and returns similar to the intraprocedural control transfers and ensures the validity of interprocedural paths by maintaining a history of calls in terms of call strings. A *call string* at a program point u is a sequence $c_1c_2\ldots c_k$ of call sites corresponding to unfinished calls at u and can be viewed as a snapshot of the call stack at u; λ denotes an empty call string. Figure 1 shows a program and its supergraph. S_p and E_p denote the start and end of procedure p while those for the main program are *Entry* and *Exit*. A call site c_i is split into a call node C_i and the corresponding return node R_i and appropriate call and return edges are added. Some call strings for this program are λ, c_1, c_1c_3, $c_1c_3c_4$, $c_1c_3c_4c_4$ etc.

Call string construction is governed by the interprocedural edges in a supergraph. Let σ be a call string reaching node m in 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 S_q, call string $\sigma \cdot c_i$ reaches S_q. For a return edge $m \rightarrow n$ where m is E_p and n is R_i, if the last call site in σ is c_i then the call string remaining after removing c_i from σ reaches R_i. This ensures that the data flow information is propagated to the correct caller.

The augmented data flow information is a pair $\langle \sigma, d \rangle$ where d is the data flow value propagated along call string σ. Note that d is modified by an intraprocedural edge only. A work list based iterative algorithm is used to perform the data flow analysis. The

process terminates when no new pair $\langle \sigma, d \rangle$ is computed; merging data flow values propagated along all call strings reaching u results in a meet-over-all-interprocedurally-valid-paths solution at u for distributive frameworks.

Since matching of call and return nodes is inherently performed in the call strings method, it ensures that all interprocedurally valid paths are traversed and invalid paths are avoided. Thus use of call strings guarantees a safe and precise solution. In non-recursive programs, since the call strings are acyclic, their number is finite and all of them are generated during analysis. However, in recursive program, new call strings are generated with every visit to a call node involved in recursion. In such cases, call strings must be restricted to a finite number using explicit criteria.

Let K be the maximum number of distinct call sites in any call chain and L be the lattice of data flow values. The full call strings method [22] requires construction of all call strings of length up to $K \times (|L|+1)^2$ for computing a safe and precise solution. Intuitively, the argument by Sharir can be explained as follows: Let a data flow value at call node C_i be v_i and the corresponding value at R_i be v_i'. Since there are $|L|+1$ values for v_i and v_i' (due to presence of a fictitious value Ω), $(|L|+1)^2$ distinct combinations are possible, for which $(|L|+1)^2$ distinct call strings are required. If c_i is in recursion, $(|L|+1)^2$ occurrences of c_i guarantee that all these call strings are generated and hence guarantee all possible computations. Since there can be K distinct call sites, call strings of length $K \times (|L|+1)^2$ ensure that all possible data flow values are computed. For separable frameworks, the prescribed length reduces to $K \times (|\widehat{L}|+1)^2$ where $\widehat{L}$ is the component lattice for an entity. For bit-vector frameworks, this length is $3 \times K$.

3 Efficiency of Call Strings Approach

This section discusses the factors affecting the efficiency of the classical full call strings method.

Orthogonality of Call Strings and Data Flow Values. Analysis of non-recursive programs constructs a finite number of call strings and the termination of analysis is governed solely by the convergence of data flow values. In recursive programs, termination of call string construction needs to be ensured explicitly. Once the termination of call strings is ensured, the usual fixed point criterion can be applied to data flow values to ensure the termination of analysis exactly as in iterative intraprocedural analysis.

In the classical full call strings method, call string construction is terminated by truncating call strings at a prescribed length. We ask the following question: Is it possible to use data flow values instead of a prescribed length to bound the cyclic call strings? Intuitively, a criterion can be devised to stop the construction of new call strings when the old values repeat along cyclic call strings. But this further raises questions regarding safety and precision: Do the call strings thus terminated ensure traversing all interprocedurally valid paths and avoiding all invalid paths? We answer these questions by characterizing the minimal set of call strings required for recursive procedures.

Issues in Terminating Call String Construction for Recursive Programs. The prescribed length defined in the classical method is based on a crude estimate to ensure

Let the cyclic call sequence be $c_x \cdot c_{x+1} \cdots c_{x+y} \cdots \equiv \sigma_c$. Let the flow function along the cyclic call sequence be f, along cyclic return sequence be g, and that along the recursion ending path be h. The prescribed length is m.

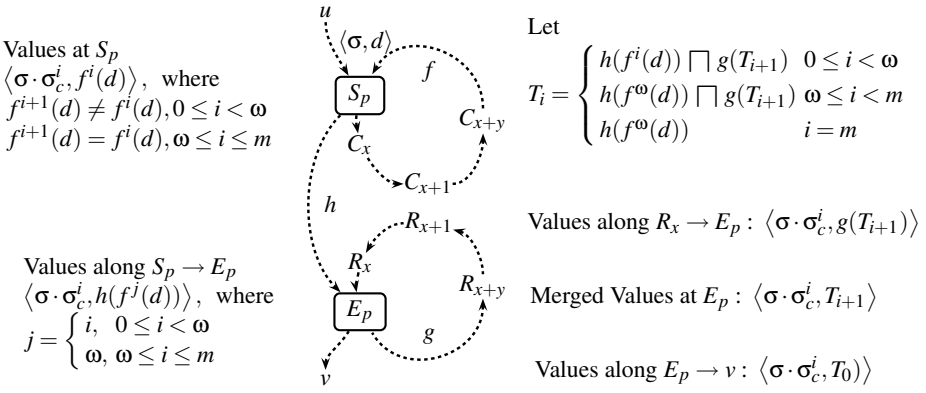

Fig. 2. Modeling recursion for call strings. σ_c may have multiple occurrence of a call node and hence can be any arbitrary recursive call sequence. Though the recursion ending path has been shown in procedure p it may not exist in p but in some other procedure in recursion.

complete analysis in both call and return sequences as explained below. Hence many cyclic call strings generated using the prescribed length are redundant in that they carry the same data flow information as some shorter call strings.

Consider the situation in Figure 2 which models a recursive call. The strongly connected component consisting of call nodes $(C_x, C_{x+1}, \ldots, C_{x+y})$ is a *cyclic call sequence* and is denoted by σ_c. The corresponding *cyclic return sequence* $(R_{x+y}, \ldots, R_{x+1}, R_x)$ forms another strongly connected component which we denote by σ_r. The dashed line from S_p to E_p represents the recursion ending control flow path. In a valid interprocedural path involving σ_c and σ_r, σ_c is traversed at least as many times as σ_r. Observe that we do not require the call sites along a cyclic call sequence to be distinct. Thus this figure models a general recursive path. We have shown the recursion ending path in procedure p but as Corollary 1 shows, it does not matter if this path exists in some other procedure in recursive call chain.

Each application of g requires traversing the cyclic return sequence once. In the process, the last occurrence of σ_c is removed from every call string. Thus, g can be applied only as many times as the maximum number of σ_c in any call string reaching the entry of E_p. Note that the application of f does not have such a requirement because the call strings are constructed rather than consumed while applying f. Achieving safety and precision in call strings method requires the following:

Precision. In any path from S_p to E_p, the number of applications of g should not exceed that of f. This is ensured by the call string construction algorithm implying that only interprocedurally valid paths are considered.

Safety. In order to guarantee safety, the call strings should be long enough to allow computation of all possible data flow values in both cyclic call and return sequences. In a cyclic call sequence this is guaranteed by constructing call strings $\sigma \cdot \sigma_c^i$, $0 \leq i \leq \omega$.

If we select m that is large enough to allow for computation of all possible values of the following recurrence then these call strings also guarantee convergence of data flow values in the corresponding cyclic return sequence.

$$T_i = \begin{cases} h(f^{\omega}(d)) \sqcap g(T_{i+1}) & \omega \leq i < m \\ h(f^{\omega}(d)) & i = m \end{cases} \quad (1)$$

Note that the computation starts from the last call string and is performed in the order: $T_m, T_{m-1}, \ldots, T_{m-\omega+1}, T_{m-\omega}$. The convergence lemma (Lemma 3) shows that this sequence follows a strictly descending chain. Let the length of this chain be η. Then m should be at least $\omega + \eta$. If $m < \omega + \eta$, then some data flow values corresponding to unbounded recursion may not be computed. Since the values of ω and η are not known a priori, the classical prescribed length subsumes the possible worst case scenarios.

4 An Efficient Variant of Call Strings Approach

This section presents the proposed variant of call strings method.

4.1 Concepts and Notations

A program point v is *context dependent* on program point u if (a) there is a path from u to v which is a subpath of an interprocedurally valid path from *Entry* to v, and (b) on every such path from u to v, every occurrence of an E_p is matched by a corresponding occurrence of S_p. For a procedure p, all program points within p and all program points within all callees in every call chain starting in p, are context dependent on S_p.

We view call and return nodes as being *significant* nodes. When v is context dependent on u, a *context defining path* from u to v is a sequence of significant nodes appearing in a path from u to v such that this path is a subpath of a valid interprocedural path from *Entry* to v. Observe that each adjacent pair of nodes in a context defining path may correspond to many intraprocedural paths. Let $Cd(u)$ denote the set of program points which are context dependent on program point u. Then, $Cdp(u,v)$ denotes the set of context defining paths from u to $v \in Cd(u)$. $Cs(u,v)$ denotes the set of call strings corresponding to paths in $Cdp(u,v)$.

The concept of context defining path can be seen as a more general abstraction of the concept of the same-level-valid-paths [19] which are interprocedural paths which start and end in the same procedure and have matching call return pairs.

Let $\mathcal{V}(\sigma,u)$ denote the value associated with call string σ at program point u. We define the equivalence of call strings at a given program point u as follows:

$$\sigma_1 \overset{u}{=} \sigma_2 \overset{def}{=} \{\sigma_1, \sigma_2\} \subseteq Cs(Entry, u) \wedge \mathcal{V}(\sigma_1, u) = \mathcal{V}(\sigma_2, u) \quad (2)$$

Equivalence of contexts in terms of data flow values has been observed by [14, 24] and has been used for non-recursive portions of programs.

We assume that the work list based analysis is *intraprocedurally eager* i.e. it processes intraprocedural paths completely before propagating data flow information from a significant node to another significant node. This requires two separate work lists: One for intraprocedural nodes and the other for significant nodes. A significant node is selected for processing only when the work list of intraprocedural nodes is empty.

4.2 Call String Invariants

This section presents the following results: The *context invariance* lemma (Lemma 1) guarantees that the same set of call strings reaches all program points in a procedure. Hence, if a mechanism is devised to ignore some call strings in a procedure, it would be possible to reconstruct them wherever they are required. The *call strings equivalence* lemma (Lemma 2) guarantees that if call strings are partitioned on the basis of data flow values, the equivalence classes remain unchanged in a procedure although the values associated with them may change. The *convergence* lemma (Lemma 3), and the *sufficiency* theorem (Theorem 1) guarantee that if there is a way of computing the correct value of $\sigma \cdot \sigma_c^{\omega}$ at E_p, call strings $\sigma \cdot \sigma^i,\ \omega < i \leq m$ need not be constructed (Figure 2).

Lemma 1. ***(Context Invariance).*** *The calling contexts of all intraprocedural program points in a procedure are identical.*
INTUITION : Calling contexts of a procedure depend on the callers so they cannot be different for different program points within the procedure.
PROOF : Omitted. ■

Lemma 2. ***(Call String Equivalence).*** *Consider $v \in \mathrm{Cd}(u)$. Assume that the recursive paths in $\mathrm{Cdp}(u,v)$ are unbounded. When the work list of intraprocedural nodes is empty in an intraprocedurally eager call strings based method,*

$$\sigma_1 \stackrel{u}{=} \sigma_2 \Rightarrow \forall \sigma \in \mathrm{Cs}(u,v),\ (\sigma_1 \cdot \sigma) \stackrel{v}{=} (\sigma_2 \cdot \sigma)$$

INTUITION : Since σ_1 and σ_2 are transformed in the same manner by following the same set of paths, the values associated with them will also be transformed in the same manner and will continue to remain equal.
PROOF : Omitted. ■

This lemma assumes unbounded recursion. However, practical call strings method uses a prescribed length. Hence as illustrated in Figure 2, last η call strings do not have the same value at E_p in spite of the fact that they have the same value at S_p. If the call strings had unbounded occurrences of σ_c, then this exception would not arise. However, this exception does not matter because the associated values follow a strictly descending chain and converge on the least value as shown by the following lemma. It refers to Section 3 and Figure 2.

Lemma 3. ***(Convergence).*** *Assume that the call strings method constructs call strings long enough so that all call strings $\sigma \cdot \sigma_c^i,\ 0 \leq i \leq m$ are constructed where $m \geq \omega + \eta$ for all possible values of ω and η. Then,*

$$\forall \eta,\ \ \mathcal{V}(\sigma \cdot \sigma_c^{m-\eta}, E_p) \sqsubseteq \mathcal{V}(\sigma \cdot \sigma_c^i, E_p),\ m - \eta \leq i \leq m$$

INTUITION : When a data flow value is repeatedly computed using the same function and is merged with the same value at each step, the resulting values must follow a strictly descending chain until convergence.

PROOF : Since call strings $\sigma \cdot \sigma_c^i, \omega \leq i \leq m$ have the same data flow value at S_p, from Lemma 2, they have the same value, say d', just before E_p along the recursion ending path. Since $\omega \leq m - \eta$, the value associated with call strings $\sigma \cdot \sigma_c^i, m - \eta \leq i \leq m$ at E_p along the recursion ending path will also be d'. From Figure 2 and equation (1),

$$\mathcal{V}(\sigma \cdot \sigma_c^i, E_p) = T_i = \begin{cases} d' \sqcap g(T_{i+1}) & m - \eta \leq i < m \\ d' & i = m \end{cases}$$

Then the proof obligation reduces to showing $T_{m-\eta} \sqsubseteq T_i, \;\; m - \eta \leq i \leq m$. We prove this by inducting on the distance of i from m by rewriting T_i as $T_{m-j}, 0 \leq j \leq \eta$ and by showing that $T_{m-(j+1)} \sqsubseteq T_{m-j}$, $0 \leq j < \eta$. The basis of induction is $j = 0$. Since $T_m = d'$ and $T_{m-1} = d' \sqcap (\ldots)$, it follows that $T_{m-1} \sqsubseteq T_m$. For the inductive step, assume that $T_{m-(j+1)} \sqsubseteq T_{m-j}$. We need to show that $T_{m-(j+2)} \sqsubseteq T_{m-(j+1)}$. From (1),

$$T_{m-(j+2)} = d' \sqcap g(T_{m-(j+1)}) \tag{3}$$

$$T_{m-(j+1)} = d' \sqcap g(T_{m-j}) \tag{4}$$

From the inductive hypothesis and monotonicity of functions,

$$T_{m-(j+1)} \sqsubseteq T_{m-j} \Rightarrow g(T_{m-(j+1)}) \sqsubseteq g(T_{m-j})$$

The inductive step follows by substituting this in the right hand sides of (3) and (4) and comparing them. ■

If the recursion ending path is not within procedure p but is in some other procedure, then T_i at E_p will simply be $g^{i-m}(d')$.

Lemma 4. ***(Convergence in a Cycle).*** *When the computation of a data flow value converges at a program point in a cycle, it must converge at each program point in the cycle. Further, due to monotonicity, all values must converge in the same direction.*

Corollary 1. *T_i of the form $g^{i-m}(d')$ at E_q must converge.*

Theorem 1. ***(Sufficiency of Cyclic Call Strings).***

$$\prod_{i=0}^{m} \mathcal{V}(\sigma \cdot \sigma_c^i, E_p) = \prod_{i=0}^{\omega} \mathcal{V}(\sigma \cdot \sigma_c^i, E_p)$$

INTUITION : When the data flow values along call strings in a cyclic return sequence follow a descending chain, only the last value matters in the overall merge.

PROOF : Since the data flow value computation converges for the value associated with $\sigma \cdot \sigma_c^{m-\eta}$, from Lemma 3, $\mathcal{V}(\sigma \cdot \sigma_c^i, E_p) = \mathcal{V}(\sigma \cdot \sigma_c^{i+1}, E_p), \; \omega \leq i < m - \eta$. As consequence, $\mathcal{V}(\sigma \cdot \sigma_c^{\omega}, E_p) \sqsubseteq \mathcal{V}(\sigma \cdot \sigma_c^i, E_p), \; \omega \leq i < m$ which proves the theorem. ■

4.3 Modifying Call Strings Method

The basic principle of our approach is to maintain a single representative call string for an equivalence class within the scope of a maximal context dependent region. For

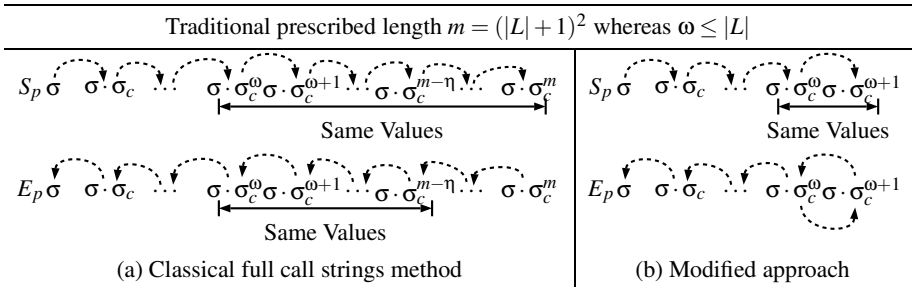

Fig. 3. Modifying the call strings method for representing and regenerating cyclic call strings.

procedure p, the decision of representation is taken at S_p and remains valid at all program points which are context dependent on S_p. E_p is the last such point and the call strings must be regenerated so that appropriate data flow values can be propagated to different callers of p. Similar to the scope of variables in a program, this representation may be "shadowed" by other context dependent regions created by procedure calls.

Let $shortest(\sigma, u)$ denote the shortest call string which has the same value as σ at u. Then, representation at S_p and regeneration at E_p is performed as follows:

$$represent(\langle \sigma, d \rangle, S_p) = \langle shortest(\sigma, S_p), d \rangle \tag{5}$$

$$regenerate(\langle \sigma, d \rangle, E_p) = \{\langle \sigma', d \rangle \mid \mathcal{V}(\sigma, S_p) = \mathcal{V}(\sigma', S_p)\} \tag{6}$$

This change obviates the need to construct all call strings up to a prescribed length. For finite lattices, the termination of call strings automatically follows. Effectively, this facilitates fixed point computation of contexts and avoids merging contexts.

Our method constructs call strings $\sigma \cdot \sigma_c^i, 0 \leq i \leq \omega$ for the recursive contexts. Call string $\sigma \cdot \sigma_c^{\omega+1}$ is represented by $\sigma \cdot \sigma_c^{\omega}$ at S_p and no subsequent call string is created. Thus, call strings $\sigma \cdot \sigma_c^i$, $\omega + 1 < i \leq m$ are not regenerated at E_p as illustrated in Figure 3. All other call strings are regenerated completely.

Observe that the actual value of ω governs the construction of call strings (without the need of knowing ω) in our method. However, the value of η does not play any role in construction of call strings. This is because the computation of $f^i(d)$ in a cyclic call sequence (Figure 2) begins with the first call string whereas the computation of T_i in the corresponding cyclic return sequence begins with the last call string.

4.4 Safety, Precision, Efficiency, and Complexity

Theorem 2. ***(Safety and Precision).*** *The final data flow values computed by representing and regenerating call strings using (5) and (6) are identical to the values computed by the original call strings method with length bound.*

INTUITION : Representation and regeneration discards only those call strings which contain redundant values and performs the desired computation iteratively.

PROOF : For the non-recursive contexts, the theorem is obvious. For recursive contexts we show that our method computes the same data flow value for call string $\sigma \cdot \sigma_c^{\omega}$ at E_p as would be computed by the original method.

At E_p, $\sigma \cdot \sigma_c^{\omega+1}$ is regenerated and the data flow value (say d') associated with $\sigma \cdot \sigma_c^{\omega}$ is propagated to it. The analysis propagates the pair $\langle \sigma \cdot \sigma_c^{\omega+1}, d' \rangle$ along the cyclic return sequence. This traversal removes the last occurrence of σ_c from $\sigma \cdot \sigma_c^{\omega+1}$, computes $g(d')$, which is merged with the value of $\sigma \cdot \sigma_c^{\omega}$ along the recursion ending path. Thus $\mathcal{V}(\sigma \cdot \sigma_c^{\omega}, E_p) = d' \sqcap g(d')$ after one traversal. This is same as the value associated with call string $\sigma \cdot \sigma_c^{m-1}$ in the original method. At E_p, this is again copied to the call string $\sigma \cdot \sigma_c^{\omega+1}$ overwriting the previous value and the pair $\langle \sigma \cdot \sigma_c^{\omega+1}, d' \sqcap g(d') \rangle$ is propagated along the cyclic return sequence. The process repeats as long as new values are computed for $\sigma \cdot \sigma_c^{\omega}$; effectively, traversal i over the cyclic return sequence computes the value T_{m-i} for $\sigma \cdot \sigma_c^{\omega}$. The process terminates after η traversals. This computes the desired value for $\sigma \cdot \sigma_c^{\omega}$. ■

Effectively, our method computes the correct value for $\sigma \cdot \sigma_c^{\omega}$ by iterating over the cyclic return sequence η times, rather than constructing all call strings up to $\sigma \cdot \sigma_c^{m}$. Traditional prescribed length m is orders of magnitude larger than ω, hence terminating the call strings construction at $\sigma \cdot \sigma_c^{\omega}$ results in a dramatic reduction in the number of call strings. Further improvements in efficiency arise because the reduction in the number of call strings is exponential—at each call site, much fewer call strings are passed on to callees along a call chain. The iterative computation does not entail any additional cost because these computations are anyway performed by the original method.

The elegance of our method lies in the fact that not only does it reduce space and time dramatically in practice, it also brings down the worst case complexity of call string length from quadratic to linear in the size of the lattice.

Theorem 3. ***(Complexity).*** *Using the value based termination of call strings, the maximum length of a call string is* $K \times (|L| + 1)$.

INTUITION : At the start of each procedure, the call strings are partitioned by the data flow values associated with them.

PROOF : The lemma trivially holds for call strings in non-recursive contexts. For recursive contexts, we maintain the call strings $\sigma \cdot \sigma_c^{\omega}$ at the exit of S_p. Since all call strings which have the same value are represented by a single call string, at most $|L|$ distinct call strings will be maintained at S_p. Thus, $\omega \leq |L|$ and no call site needs to appear more than $|L|$ times in a call string. We may have an additional call string at the entry of S_p which gets represented at exit of S_p. Hence the theorem. ■

Even in the worst case, our method would construct much fewer call strings. Further, in practice, our method does not construct all call strings up to the worst case length. This is different from the original method which requires construction of all call strings of length up to $K \times (|L| + 1)^2$.

Corollary 2. *For separable frameworks, the bound reduces to* $K \times (|\widehat{L}| + 1)$ *where* $\widehat{L}$ *is the component lattice representing the data flow values of one entity. For bit vector frameworks, it further reduces to* $K \times 3$ *since* $|\widehat{L}| = 2$.

4.5 An Example of Points-To Analysis

Consider the supergraph in Figure 1 for interprocedural May Points-to analysis [5, 11]. Figure 4 shows some important steps in the analysis using our method; In_n and

Out_n denote entry and exit points of n. The data flow information is stored as $\langle \sigma, d \rangle$ where d is the May points-to information which is a set of elements $x \triangleright S$ indicating that x points to the variables contained in set S.

Observe the computation of representative call strings at node S_p as shown in rows 5 and 6. Since both call strings c_1 and c_2 reaching the entry of procedure p carry the same data flow value, they are represented by a single call strings c_1. Note that c_2 is also eligible as the representative call string. Further, the *represent* function is applied at S_q (see rows $10, 11$) where two call strings c_1c_3 and $c_1c_3c_4$ carry the same data flow value and hence are represented by the shortest call string c_1c_3.

The regeneration takes place at the exit of procedure q (see rows $12, 13$). The regenerated call string $c_1c_3c_4$ reaches R_4. Effect of statement $x = *x$ in node n_4 is observed on the data flow value associated with call string c_1c_3. At In_{E_q}, values associated with c_1c_3 are merged (row 16) and function *regenerate* is applied once again (row 17). In the subsequent visit to node n_4 (not shown in the table), statement $x = *x$ modifies the points-to information of x again and merging of information and regeneration of call strings is performed once again at E_q.

Eventually, call string c_1c_3 reaches R_3 and is transformed into c_1. This call string reaches E_p and function *regenerate* is applied to reconstruct call strings c_1 and c_2 at the exit of E_p as shown in rows $18, 19$ of Figure 4. Effectively, we perform safe and precise May points-to analysis using only acyclic call strings. We construct 5 call strings for the same. The overall lattice of May points-to framework for this example contains 512 elements. Considering $K = 3$ (the total number of distinct call sites in a call chain), the classical method would construct all call strings with lengths up to $7,89,507$. Clearly, it would require millions of call strings.

4.6 An Approximate Version

It is possible to increase the efficiency of the proposed method by using an approximate version which can adjust the approximation on demand. The approximation is quantified in terms of the number of occurrences of a call site in any call string. Let this number be δ. When a call string σ containing $\delta - 1$ occurrences of call site c_i reaches call node C_i, $\sigma \cdot c_i$ is created. If some other call string σ' containing $\delta - 1$ occurrences of c_i reaches C_i, instead of constructing $\sigma' \cdot c_i$ the value of σ' is merged with $\sigma \cdot c_i$. In other words, the first call string that grows to contain δ occurrences of c_i becomes the representative call string for all call strings containing δ or more occurrences of c_i. When a call string with the prefix $\sigma \cdot c_i$ reaches C_i, it is represented by $\sigma \cdot c_i$ (which is the representative call string) instead of suffixing another c_i to it and its modified value is merged with the earlier value of $\sigma \cdot c_i$ at C_i. The process is repeated iteratively until the merged value converges. This converged value is then propagated back to each represented call string during regeneration at R_i. Since no context is missed out, this is safe but since values are merged across contexts, this is possibly imprecise. The degree of imprecision depends on the choice of δ. The existing methods which merge the values in recursive contexts can be seen as a special case of our approximate method with $\delta = 1$.

Apart from increasing efficiency, demand driven summarization facilitates application of call strings method to data flow frameworks with infinite lattices which have finite heights (eg. constant propagation [1]).

	Point i	New information at i		Point i	New information at i
1	In_{Entry}	$\langle \lambda, \{x \triangleright \emptyset, y \triangleright \emptyset, z \triangleright \emptyset\}\rangle$	11	Out_{S_q}	$\langle c_1c_3, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
...			...		
2	Out_{n_2}	$\langle \lambda, \{x \triangleright \{y\}, y \triangleright \emptyset, z \triangleright \{x\}\}\rangle$	12	In_{E_q}	$\langle c_1c_3, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
...			13	Out_{E_q}	$\langle c_1c_3, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$,
3	Out_{C_1}	$\langle c_1, \{x \triangleright \{y\}, y \triangleright \emptyset, z \triangleright \{x\}\}\rangle$			$\langle c_1c_3c_4, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
4	Out_{C_2}	$\langle c_2, \{x \triangleright \{y\}, y \triangleright \emptyset, z \triangleright \{x\}\}\rangle$	14	In_{R_4}	$\langle c_1c_3c_4, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
5	In_{S_p}	$\langle c_1, \{x \triangleright \{y\}, y \triangleright \emptyset, z \triangleright \{x\}\}\rangle$,	...		
		$\langle c_2, \{x \triangleright \{y\}, y \triangleright \emptyset, z \triangleright \{x\}\}\rangle$	15	Out_{n_5}	$\langle c_1c_3, \{x \triangleright \{z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
6	Out_{S_p}	$\langle c_1, \{x \triangleright \{y\}, y \triangleright \emptyset, z \triangleright \{x\}\}\rangle$	16	In_{E_q}	$\langle c_1c_3, \{x \triangleright \{y,z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
...			17	Out_{E_q}	$\langle c_1c_3, \{x \triangleright \{y,z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$,
7	Out_{C_3}	$\langle c_1c_3, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$			$\langle c_1c_3c_4, \{x \triangleright \{y,z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
...			...		
8	In_{S_q}	$\langle c_1c_3, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$	18	In_{E_p}	$\langle c_1, \{x \triangleright \{x,y,z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
...			19	Out_{E_p}	$\langle c_1, \{x \triangleright \{x,y,z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$,
9	Out_{C_4}	$\langle c_1c_3c_4, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$			$\langle c_2, \{x \triangleright \{x,y,z\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$
...			...		
10	In_{S_q}	$\langle c_1c_3, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$,			
		$\langle c_1c_3c_4, \{x \triangleright \{y\}, y \triangleright \{z\}, z \triangleright \{x\}\}\rangle$			

Fig. 4. Some important steps in intraprocedurally eager work list algorithm for interprocedural May points-to analysis using value based termination of call strings for supergraph in Figure 1.

5 Related Work

We compare our work with other methods on the basis of precision, efficiency, generality and simplicity. The approximate call strings method [22] which retains fixed length suffixes is a popular variant of call strings method. Although it is efficient and flexible, it compromises on precision in recursive as well as non-recursive programs and the degree of precision varies with the length of suffixes.

Functional approach [22] involves computing flow functions in a context independent manner and applying them in a context sensitive manner. Although this approach guarantees precision, it is known to be inefficient due to high time and space complexity resulting from function computations [1]. Tabulation method [22] is an efficient implementation of functional approach, which uses memoization to store input and output data flow values at each program point, instead of storing the functions. Similar to our approach, this approach also uses the basic principle of restricting the reanalysis of procedures only for distinct inputs. However, unlike our method, tabulation method merges the newly computed data flow values with the old values at each program point to guarantee termination. Further, since contexts are not remembered separately, meaningful approximation is not possible and hence it cannot be used for frameworks with infinite lattices. The method of computing partial transfer functions (PTF) [24, 18] looks very similar to tabulation. However, PTFs involve summarization of input in recursive contexts whereas our method and tabulation do not do so and hence are more precise. The graph reachability method [19, 21, 10] is a variant of tabulation based functional

approach which requires computation of an exploded supergraph. It is applicable only to finite distributive frameworks.

Many approaches have been developed specifically for context-sensitive points-to analysis. BDD-based approaches [23, 25, 26] construct all acyclic contexts but merge values along recursive portions resulting in loss of precision. Since BDDs have efficient implementations and they exploit the commonality across contexts carrying equivalent values [14], these approaches are scalable. Many approaches [4, 13, 15, 7, 16] achieve efficiency by using flow-insensitive algorithms for intraprocedural analysis thereby causing additional imprecision. The context-sensitive points-to analysis using invocation graph [5] requires construction of separate invocation graph and is reported not to be scalable [23]. This method computes conservative solution along recursive portions. Summary-based points-to analysis approaches [24] are reported to be the precise, but they do not guarantee full precision along recursive portions. As observed in a comparison of context sensitive points-to analyses [14], treating recursive portions in a context insensitive manner leads to significant imprecision in practical programs.

Some context sensitive methods (eg. automata based methods [6, 20, 3], generic assertion based method [8], linear algebra based method [17]) have approached interprocedural data flow analysis from a view point of building theoretical underpinnings and their precision-efficiency trade off or generality (eg. applicability to frameworks such as points-to analysis) is not clear.

We feel that context-insensitivity along recursive paths is being looked upon as an unavoidable compromise for efficiency and is being accepted as a regular practice [9]. This may be because the orthogonality of bounding contexts and computing data flow values makes it impossible to identify and eliminate all redundant contexts. To ensure precision, the only available option is to use functional approaches or to use the worst case bounds for call strings. Both these approaches are extremely inefficient.

The occurrence based bound for call strings for bit-vector frameworks [12] is an improvement over the classical length bound [22]. It constructs call strings with any call site occurring at most 3 times instead of all call strings with lengths up to $3K$. However, it still allows many redundant call strings since the termination of call strings is orthogonal to the convergence of data flow values.

6 Empirical Measurements

We have implemented interprocedural Reaching Definitions analysis using the proposed algorithm in gcc 4.0 as an additional pass that constructs supergraph and performs the call strings based analysis on the Gimple IR. We have measured the performance of the algorithm on the following programs: `Hanoi`[1], `sim`[2], `bit_gray`[3], `181.mcf` and `256.bzip2` from SPEC-2000, `analyzer, distray, mason` and `fourinarow` from FreeBench v1.03 suite. Among these programs, `analyzer, distray` and `256.bzip2` are non-recursive whereas all other programs are recursive. These experiments were carried out on a P4 (3.06 GHz) machine with 1GB RAM running Fedora Core 6.

[1] `http://www.ece.cmu.edu/~ece548/hw/lab1/hanoi.c`

[2] `http://gd.tuwien.ac.at/perf/benchmark/aburto/sim/sim.c`

[3] `http://paul.rutgers.edu/~rhoads/Code/bit_gray.c`

Program	LoC	#*F*	#*C*	3*K* length bound					Proposed Approach			
				K	#CS	MaxL	#CSPN	Time	#CS	MaxL	#CSPN	Time
`hanoi`	33	2	4	4	100000+	12	99922	3973×10^3	8	3	7	2.37
`bit_gray`	53	5	11	7	100000+	21	31374	2705×10^3	17	4	6	3.83
`analyzer`	288	14	20	2	21	2	4	20.33	21	2	4	1.39
`distray`	331	9	21	6	96	6	28	322.41	22	3	4	1.11
`mason`	350	9	13	8	100000+	11	22143	432×10^3	14	3	4	0.43
`fourinarow`	676	17	45	5	510	15	158	397.76	46	3	7	1.86
`sim`	1146	13	45	8	100000+	14	33546	1427×10^3	211	13	105	234.16
`181_mcf`	1299	17	24	6	32789	18	32767	484×10^3	41	9	11	5.15
`256_bzip2`	3320	63	198	7	492	7	63	258.33	406	7	34	200.19

LoC is the number of lines of code, #*F* is the number of procedures, #*C* is the number of call sites, #*CS* is the number of call strings (100000+ indicates that call strings construction was aborted after 100000 call strings), #*CSPN* denotes the maximum number of call strings reaching any node. MaxL denotes the maximum length of any call strings. The analysis time is in milliseconds.

Fig. 5. Empirical measurements

Figure 5 gives the details of the benchmark programs and the call string related measurements for the $3 \times K$ length [22] and the proposed method. For the purpose of experimentation we had to restrict the number of call strings to 10^5 for $3 \times K$ bound. This was done primarily due to the compiler running out of space. The table clearly shows that our approach of terminating call strings construction using data flow values reduces the number of call strings and hence the analysis time by orders of magnitude.

7 Conclusions and Future Work

The classical full call strings method is context sensitive and computes as precise solution as is statically possible. However, it suffers from terrible inefficiency and hence has been relegated to the set of classical methods which are of academic interest only. This paper resurrects and rejuvenates the call strings method by observing some subtle insights and proposing minimal changes to the method. These changes are simple, do not impose any additional constraints, and faithfully retain the essential principles of the method and the consequent properties: precision, simplicity, and generality. These changes discard call strings where they are not required, regenerate them where they are required and iteratively compute data flow values in cyclic call strings in return sequences as summarized in Figure 3. This results in dramatic improvements in efficiency.

Our investigations deviate from the current trends along the following two aspects:

– Most contemporary investigations seem to assume that compromising precision (at least in recursive contexts) is essential for achieving efficiency. We believe that any trade-off between precision and efficiency without making a clear distinction between relevant contexts and irrelevant contexts is undesirable. We have shown that this distinction can be very easily and efficiently made by using the convergence of data flow values for convergence of contexts without compromising on precision.

– A majority of contemporary investigations involve specialized algorithms in order to achieve efficiency. They may be specialized in terms of a very sophisticated representation of the programs or in terms of using insights from the specific analyses for which they are implemented. We believe that it is important to seek efficiency in a general method which is applicable to all data flow frameworks (including those with infinite lattices) and which can be implemented very easily. Simplicity and generality are essential for exploring the possibility of automatic construction of interprocedural data flow analyzers. We find this direction to be promising because the scalability of our method depends on the convergence of data flow values rather than merely on program structure. When programs are written in modular fashion with loose coupling between different modules, the convergence of data flow values does not scale with program size as much as the number of contexts.

We have implemented this method for Reaching Definitions analysis and the results are very promising. We are in the process of implementing this method for points-to analysis and would like to test the method on large programs. Note that point-to analysis is non-distributive and the classical call string method would also suffer from imprecision. Our variant does not create any additional imprecision because the results presented in this paper do not assume distributivity property.

Our quick and dirty implementation was aimed at the first level measurements. We would like to improve the implementation by engineering better data structures and algorithms and observe their impact on the efficiency. We would also like to measure the precision vs. efficiency trade-off using the approximate version of our method.

Acknowledgments

Implementation of these analyses was carried out by Seema Ravandale. Divya Krishan was involved in the implementation of earlier versions of call strings methods.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.
2. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, pages 33–50, September 1995.
3. P. Amiranoff, A. Cohen, and P. Feautrier. Beyond iteration vectors: Instancewise relational abstract domains. In *Static Analysis Symposium*, pages 161–180, 2006.
4. M. Burke, P. Carini, J. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the pressence of pointers. In *Lecture Notes in Computer Science, 892*. 1995.
5. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
6. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Foundations of Software Science and Computation Structure*, pages 14–30, 1999.
7. M. Fahndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

8. S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *The 16th European Symposium on Programming*. Springer, March 2007.
9. B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.
10. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *3rd ACM Symposium on Foundations of Software Engineering*, pages 104–115, 1995.
11. A. Kanade, U. P. Khedker, and A. Sanyal. Heterogeneous fixed points with application to points-to analysis. In *Proc. of the Asian Symposium on Programming Languages and Systems*, pages 298–314, 2005.
12. B. Karkare and U. P. Khedker. An improved bound for call-strings based interprocedural analysis of bit vector frameworks. *ACM Trans. Program. Lang. Syst.*, 29(6):38, 2007.
13. C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.
14. O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proc. of the International Conference on Compiler Construction*, pages 47–64, March 2006.
15. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. *SIGSOFT Software Engineering Notes*, 24(6):199–215, 1999.
16. A. Milanova. Light context-sensitive points-to analysis for java. In *Proc. of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2007.
17. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 330–341, New York, NY, USA, 2004.
18. B. R. Murphy and M. S. Lam. Program analysis with partial transfer functions. In *Proc. of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 94–103, 2000.
19. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.
20. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, 2005.
21. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
22. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis : Theory and Applications*. Prentice-Hall Inc., 1981.
23. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, June 2004.
24. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.
25. J. Zhu. Towards scalable flow and context sensitive pointer analysis. In *Proc. of the 42nd Annual Conference on Design Automation*, pages 831–836, 2005.
26. J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–157, 2004.