# Large Spurious Cycle in Global Static Analyses and Its Algorithmic Mitigation

Hakjoo Oh

School of Computer Science and Engineering
Seoul National University

**Abstract.** We present a simple algorithmic extension of the classical call-strings approach to mitigate substantial performance degradation caused by spurious interprocedural cycles. Spurious interprocedural cycles are, in a *realistic* setting, key reasons for why approximate call-return semantics in both context-sensitive and -insensitive static analysis can make the analysis much slower than expected.

In the traditional call-strings-based context-sensitive static analysis, because the number of distinguished contexts must be finite, multiple call-contexts are inevitably joined at the entry of a procedure and the output at the exit is propagated to multiple return-sites. We found that these multiple returns frequently create a single large cycle (we call it "butterfly cycle") covering almost all parts of the program and such a spurious cycle makes analyses very slow and inaccurate.

Our simple algorithmic technique (within the fixpoint iteration algorithm) identifies and prunes these spurious interprocedural flows. The technique's effectiveness is proven by experiments with a realistic C analyzer to reduce the analysis time by 7%-96%. Since the technique is *algorithmic*, it can be easily applicable to existing analyses without changing the underlying abstract semantics, it is orthogonal to the underlying abstract semantics' context-sensitivity, and its correctness is obvious.

## 1 Introduction

In a global semantic-based static analysis, it is inevitable to follow some spurious (unrealizable or invalid) return paths. Even when the underlying abstract semantics is context-sensitive, because the number of distinguished contexts must be finite, multiple call-contexts are joined at the entry of a procedure and the output at the exit are propagated to multiple return-sites. For example, in a conventional way of avoiding invalid return paths by distinguishing a finite $k \geq 0$ call-sites to each procedure, the analysis is doomed to still follow spurious paths if the input program's nested call-depth is larger than the $k$. Increasing the $k$ to remove more spurious paths quickly hits a limit in practice because of the increasing analysis cost in memory and time.

In this article we present the following:

- in a realistic setting, these multiple returns often create a single large flow cycle (we call it "butterfly cycle") covering almost all parts of the program,

Z. Hu (Ed.): APLAS 2009, LNCS 5904, pp. 14–29, 2009.

– such big spurious cycles make the conventional call-strings method that distinguishes the last $k$ call-sites [14] very slow and inaccurate,
– this performance problem can be relieved by a simple extension of the call-strings method,
– our extension is an algorithmic technique within the worklist-based fixpoint iteration routine, without redesigning the underlying abstract semantics, and
– the algorithmic technique works regardless of the underlying abstract semantics' context-sensitivity. The technique consistently saves the analysis time, without sacrificing (or with even improving) the analysis precision.

### 1.1 Problem: Large Performance Degradation By Inevitable, Spurious Interprocedural Cycles

Static analysis' inevitable spurious paths make spurious cycles across procedure boundaries in global analysis. For example, consider the semantic equations in Fig 1 that (context-insensitively ($k > 0$)) abstract two consecutive calls to a procedure. The system of equations says to evaluate equation (4) and (6) for every return-site after analyzing the called procedure body (equation (3)). Thus, solving the equations follows a cycle: (2) $\rightarrow$ (3) $\rightarrow$ (4) $\rightarrow$ (5) $\rightarrow$ (2) $\rightarrow \cdots$.

Such spurious cycles degrade the analysis performance both in precision and speed. Spurious cycles exacerbate the analysis imprecision because they model spurious information flow. Spurious cycles degrade the analysis speed too because solving cyclic equations repeatedly applies the equations in vain until a fixpoint is reached.

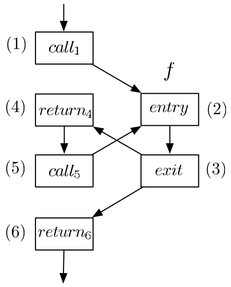
The performance degradation becomes dramatic when the involved interprocedural spurious cycles cover a large part of the input program. This is indeed the case in reality. In analyzing real C programs, we observed that the analysis follows (Section 2) a single large cycle that spans almost all parts of the input program. Such spurious cycles size can also be estimated by just measuring the strongly connected components (scc) in the "lexical"[1] control flow graphs. Table 1 shows the sizes of the largest scc in some open-source programs.[2] In most programs, such cycles cover most (80-90%) parts of the programs. Hence, globally analyzing a program is likely to compute a fixpoint of a function that describes almost all parts of the input program. Even when we do context-sensitive analysis ($k > 0$), large spurious cycles are likely to remain (Section 2).

### 1.2 Solution: An Algorithmic Mitigation without Redesigning Abstract Semantics

We present a simple algorithmic technique inside a worklist-based fixpoint iteration procedure that, without redesigning the abstract semantics part, can

[1] One node per lexical entity, ignoring function pointers.

[2] We measured the sizes of all possible cycles in the flow graphs. Note that interprocedural cycles happen because of either spurious returns or recursive calls. Because recursive calls in the test C programs are immediate or spans only a small number of procedures, large interprocedural cycles are likely to be spurious ones.



$$\begin{array}{lll}
call_1 & = \cdots & \cdots\ (1) \\
entry & = call_1 \sqcup call_5 & \cdots\ (2) \\
exit & = \hat{\mathcal{F}}_{body_f}(entry) & \cdots\ (3) \\
return_4 & = exit & \cdots\ (4) \\
call_5 & = return_4 & \cdots\ (5) \\
return_6 & = exit & \cdots\ (6)
\end{array}$$

**Fig. 1.** Spurious cycles because of abstract procedure calls and returns. The right-hand side is a system of equations and the left-hand side shows the dependences between the equations. Note a dependence cycle (2) → (3) → (4) → (5) → (2) → ⋯.

effectively relieve the performance degradation caused by spurious interprocedural cycles in both context-sensitive ($k > 0$) and -insensitive ($k = 0$) analysis.

While solving flow equations, the algorithmic technique simply forces procedures to return to their corresponding called site, in order not to follow the last edge (edge (3) → (4) in Fig 1) of the "butterfly" cycles. In order to enforce this, we control the equation-solving orders so that each called procedure is analyzed exclusively for its one particular call-site. To be safe, we apply our algorithm to only non-recursive procedures.

Consider the equation system in Fig 1 again and think of a middle of the analysis (equation-solving) sequence, $\cdots \rightarrow (5) \rightarrow (2) \rightarrow (3)$, which indicates that the analysis of procedure $f$ is invoked from (5) and is now finished. After the evaluation of (3), a classical worklist algorithm inserts all the equations, (4) and (6), that depend on (3). But, if we remember the fact that $f$ has been invoked from (5) and the other call-site (1) has not invoked the procedure until the analysis of $f$ finishes, we can know that continuing with (4) is useless, because the current analysis of $f$ is only related to (5), but not to other calls like (1). So, we process only (6), pruning the spurious sequence $(3) \rightarrow (4) \rightarrow \cdots$.

We integrated the algorithm inside an industrialized abstract-interpretation-based C static analyzer [5,6] and measured performance gains derived from avoiding spurious cycles. We have saved 7%-96% of the analysis time for context-insensitive or -sensitive global analysis for open-source benchmarks.

### 1.3 Contributions

- We present a simple extension of the classical call-strings approach, which effectively reduces the inefficiency caused by large, inevitable, spurious interprocedural cycles.

  We prove the effectiveness of the technique by experiments with an industrial-strength C static analyzer [5,6] in globally analyzing medium-scale open-source programs.

**Table 1.** The sizes of the largest strongly-connected components in the "lexical" control flow graphs of real C programs. In most cases, most procedures and nodes in program belong to a single cycle.

| Program | Procedures in the largest cycle | Basic-blocks in the largest cycle |
|---|---:|---:|
| spell-1.0 | 24/31(77%) | 751/782(95%) |
| gzip-1.2.4a | 100/135(74%) | 5,988/6,271(95%) |
| sed-4.0.8 | 230/294(78%) | 14,559/14,976(97%) |
| tar-1.13 | 205/222(92%) | 10,194/10,800(94%) |
| wget-1.9 | 346/434(80%) | 15,249/16,544(92%) |
| bison-1.875 | 410/832(49%) | 12,558/18,110(69%) |
| proftpd-1.3.1 | 940/1,096(85%) | 35,386/41,062(86%) |

- The technique is meaningful in two ways. Firstly, the technique aims to alleviate one major reason (spurious interprocedural cycles) for substantial inefficiency in global static analysis.

  Secondly, it is purely an algorithmic technique inside the worklist-based fixpoint iteration routine. So, it can be directly applicable without changing the analysis' underlying abstract semantics, regardless of whether the semantics is context-sensitive or not. The technique's correctness is obvious enough to avoid the burden of a safety proof that would be needed if we newly designed the abstract semantics.
- We report one key reason (spurious interprocedural cycles) for why less accurate context-sensitivity actually makes the analyses very slow. Though it is well-known folklore that less precise analysis does not always have less cost [10,12], there haven't been realistic experiments about the explicit reason.

### 1.4 Related Work

We compare, on the basis of their applicability to general semantic-based static analyzers[3], our method with other approaches that eliminate invalid paths.

The classical call-strings approach that retains the last $k$ call-sites [14,1,9,10] is popular in practice but its precision is not enough to mitigate large spurious cycles. This $k$-limiting method is widely used in practice [1,9,10] and actually it is one of very few options available for semantic-based global static analysis that uses infinite domains and non-distributive flow functions (e.g., [1,6]). The $k$-limiting method induces a large spurious cycle because it permits multiple returns of procedures. Our algorithm is an extension of the $k$-limiting method and adds extra precision that relieves the performance problem from spurious interprocedural cycles.

[3] For example, such analyzers include octagon-based analyzers (e.g.,[2]), interval-based analyzers (e.g.,[5,6]), value set analysis [1], and program analyzer generators (e.g, [9]), which usually use infinite (height) domains and non-distributive flow functions.

Another approximate call-strings method that uses full context-sensitivity for non-recursive procedures and treats recursive call cycles as gotos is practical for points-to analysis [15,16] but, the method is too costly for more general semantic-based analysis. Though these approaches are more precise than $k$-limiting method, it is unknown whether the BDD-based method [16] or regular-reachability [15] are also applicable in practice to general semantic-based analyzers rather than pointer analysis. Our algorithm can be useful for analyses for which these approaches hit a limit in practice and $k$-limiting is required.

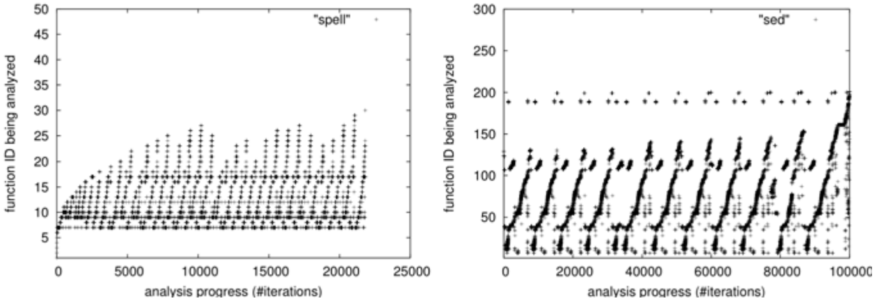
Full call-strings approaches [14,7,8] and functional approaches [14] do not suffer from spurious cycles but are limited to restricted classes of data flow analysis problems. The original full call-strings method [14] prescribes the domain to be finite and its improved algorithms [7,8] are also limited to bit-vector problems or finite domains. Khedker et al.'s algorithm [8] supports infinite domains only for demand-driven analysis. The purely functional approach [14] requires compact representations of flow functions. The iterative (functional) approach [14] requires the domain to be finite.

Reps et al.'s algorithms [11,13] to avoid unrealizable paths are limited to analysis problems that can be expressed only in their graph reachability framework. Their algorithm cannot handle prevalent yet non-distributive analyses. For example, our analyzer that uses the interval domain with non-distributive flow functions does not fall into either their IFDS [11] or IDE [13] problems. Meanwhile, our algorithm is independent of the underlying abstract semantic functions. The regular-reachability [15], which is a restricted version of Reps et al.'s algorithm [11], also requires the analysis problem to be expressed in graph reachability problem.

Chambers et al.'s method [4] is similar to ours but entails a relatively large change to an existing worklist order. Their algorithm analyzes each procedure intraprocedurally, and at call-sites continues the analysis of the callee. It returns to analyze the nodes of the caller only after finishing the analysis of the callee. Our worklist prioritizes the callee only over the call nodes that invoke the callee, not the entire caller, which is a relatively smaller change than Chambers et al.'s method. In addition, they assume worst case results for recursive calls, but we do not degrade the analysis precision for recursive calls.

## 2 Performance Problems Due to Large Spurious Cycles

If a spurious cycle is created by multiple calls to a procedure $f$, then all the procedures that are reachable from $f$ or that reach $f$ via the call-graph belong to the cycle because of call and return flows. For example, consider a call-chain $\cdots f_1 \rightarrow f_2 \cdots$. If $f_1$ calls $f_2$ multiple times, creating a spurious butterfly cycle $f_1 \bowtie f_2$ between them, then fixpoint-solving the cycle involves all the nodes of procedures that reach $f_1$ or that are reachable from $f_2$. This situation is common in C programs. For example, in GNU software, the `xmalloc` procedure, which is in charge of memory allocation, is called from many other procedures, and hence generates a butterfly cycle. Then every procedure that reaches `xmalloc` via the call-graph is trapped into a fixpoint cycle.



spell-1.0 (total #procs:31)
(2,213 LOC, > 30 repetitions)

sed-4.0.8 (total #procs:294)
(26,807 LOC, > 150 repetitions)

**Fig. 2.** Analysis localities. Because of butterfly cycles, similar patterns are repeated several times during the analysis and each pattern contains almost all parts of the programs.

In conventional context-sensitive analysis that distinguishes the last $k$ call-sites [14], if there are call-chains of length $l$ ($> k$) in programs, it's still possible to have a spurious cycle created during the first $l - k$ calls. This spurious cycle traps the last $k$ procedures into a fixpoint cycle by the above reason.

One spurious cycle in a real C program can trap as many as 80-90% of basic blocks of the program into a fixpoint cycle. Fig 2 shows this phenomenon. In the figures, the x-axis represents the execution time of the analysis and the y-axis represents the procedure name, which is mapped to unique integers. During the analysis, we draw the graph by plotting the point $(t, f)$ if the analysis' worklist algorithm visits a node of procedure $f$ at the time $t$. For brevity, the graph for sed-4.0.8 is shown only up to 100,000 iterations among more than 3,000,000 total iterations. From the results, we first observe that similar patterns are repeated and each pattern contains almost all procedures in the program. And we find that there are much more repetitions in the case of a large program (sed-4.0.8, 26,807 LOC) than a small one (spell-1.0, 2,213 LOC): more than 150 repeated iterations were required to analyze sed-4.0.8 whereas spell-1.0 needed about 30 repetitions.

## 3 Our Algorithmic Mitigation Technique

In this section, we describe our algorithmic technique. We first describe the traditional call-strings-based analysis algorithm (section 3.1). Then we present our algorithmic extension of the classical algorithm (section 3.2).

We assume that a program is represented by a supergraph [11]. A supergraph consists of control flow graphs of procedures with interprocedural edges connecting each call-site to its callee. Each node $n \in Node$ in the graph has one of the five types :

$$entry_f \mid exit_f \mid call_f^{g,r} \mid rtn_f^c \mid cmd_f$$

The subscript $f$ of each node represents the procedure name enclosing the node. $entry_f$ and $exit_f$ are entry and exit nodes of procedure $f$. A call-site in a program is represented by a call node and its corresponding return node. A call node $call_f^{g,r}$ indicates that it invokes a procedure $g$ and its corresponding return node is $r$. We assume that function pointers are resolved (before the analysis). Node $rtn_f^c$ represents a return node in $f$ whose corresponding call node is $c$. Node $cmd_f$ represents a general command statement. Edges are assembled by a function, $\mathsf{succof}$, which maps each node to its successors. *CallNode* is the set of call nodes in a program.

### 3.1 $\mathsf{Normal}_k$: A Normal Call-Strings-Based Analysis Algorithm

Call-strings are sequences of call nodes. To make them finite, we only consider call-strings of length at most $k$ for some fixed integer $k \geq 0$. We write $CallNode^{\leq k} \overset{\text{let}}{=} \Delta$ for the set of call-strings of length $\leq k$. We write $[c_1, c_2, \cdots, c_i]$ for a call-string of call sequence $c_1, c_2, \cdots, c_i$. Given a call-string $\delta$ and a call node $c$, $[\delta, c]$ denotes a call-string obtained by appending $c$ to $\delta$. In the case of context-insensitive analysis ($k = 0$), we use $\Delta = \{\epsilon\}$, where the empty call-string $\epsilon$ means no context-information.

Fig 3.(a) shows the worklist-based fixpoint iteration algorithm that performs call-strings($\Delta$)-based context-sensitive (or insensitive, when $k = 0$) analysis. The algorithm computes a table $\mathcal{T} \in Node \rightarrow State$ which associates each node with its input state $State = \Delta \rightarrow Mem$, where *Mem* denotes abstract memory, which is a map from program variables to abstract values. That is, call-strings are tagged to the abstract memories and are used to distinguish the memories propagated along different interprocedural paths, to a limited extent (the last $k$ call-sites). The worklist $\mathcal{W}$ consists of node and call-string pairs. The algorithm chooses a work-item $(n, \delta) \in Node \times \Delta$ from the worklist and evaluates the node $n$ with the flow function $\hat{\mathcal{F}}$. Next work-items to be inserted into the worklist are defined by the function $\mathcal{N} \in Node \times \Delta \rightarrow 2^{Node \times \Delta}$ :

$$\mathcal{N}(n, \delta) = \begin{cases} \{(r, \delta') \mid \delta = \lceil \delta', call_f^{g,r} \rceil_k \wedge \delta' \in \mathsf{dom}(\mathcal{T}(call_f^{g,r}))\} & \text{if } n = exit_g \\ \{(entry_g, \lceil \delta, n \rceil_k))\} & \text{if } n = call_f^{g,r} \\ \{(n', \delta) \mid n' \in \mathsf{succof}(n)\} & \text{otherwise} \end{cases}$$

where $\mathsf{dom}(f)$ denotes the domain of map $f$ and $\lceil \delta, c \rceil_k$ denotes the call-string $[\delta, c]$ but possibly truncated so as to keep at most the last $k$ call-sites.

The algorithm can follow spurious return paths if the input program's nested call-depth is larger than the $k$. The mapping $\delta'$ to $\lceil \delta', call_f^{g,r} \rceil_k$ is not one-to-one and $\mathcal{N}$ possibly returns many work-items at an exit node.

We call the algorithm $\mathsf{Normal}_k(k = 0, 1, 2, \ldots)$. $\mathsf{Normal}_0$ performs context-insensitive analysis, $\mathsf{Normal}_1$ performs context-sensitive analysis that distinguishes the last 1 call-site, and so on.

### 3.2 $\mathsf{Normal}_k/\mathsf{RSS}$: Our Algorithm

**Definition 1.** *When a procedure $g$ is called from a call node $call_f^{g,r}$ under context $\delta$, we say that $(call_f^{g,r}, \delta)$ is the call-context for that procedure call. Since* *each call node $call_f^{g,r}$ has a unique return node, we interchangeably write $(r, \delta)$ and $(call_f^{g,r}, \delta)$ for the same call-context.*

Our return-site-sensitive (RSS) technique is simple. When calling a procedure at a call-site, the call-context for that call is remembered until the procedure returns. The bookkeeping cost is limited to only one memory entry per procedure. This is possible by the following strategies:

1. **Single return:** Whenever the analysis of a procedure $g$ is started from a call node $call_f^{g,r}$ in $f$ under call-string $\delta$, the algorithm remembers its call-context $(r, \delta)$, consisting of the corresponding return node $r$ and the call-string $\delta$. And upon finishing analyzing $g$'s body, after evaluating $exit_g$, the algorithm inserts only the remembered return node and its call-string $(r, \delta)$ into the worklist. Multiple returns are avoided. For correctness, this single return should be allowed only when other call nodes that call $g$ are not analyzed until the analysis of $g$ from $(call_f^{g,r}, \delta)$ completes.
2. **One call per procedure, exclusively:** We implement the single return policy by using one memory entry per procedure to remember the call-context. This is possible if we can analyze each called procedure exclusively for its one particular call-context. If a procedure is being analyzed from a call node $c$ with a call-string $\delta$, processing all the other calls that call the same procedure should wait until the analysis of the procedure from $(c, \delta)$ is completely finished. This one-exclusive-call-per-procedure policy is enforced by not selecting from the worklist other call nodes that (directly or transitively) call the procedures that are currently being analyzed.
3. **Recursion handling:** The algorithm gives up the single return policy for recursive procedures. This is because we cannot finish analyzing recursive procedure body without considering other calls (recursive calls) in it. Recursive procedures are handled in the same way as the normal worklist algorithm.

The algorithm does not follow spurious return paths regardless of the program's nested call-depth. While $\mathsf{Normal}_k$ starts losing its power when a call chain's length is larger than $k$, $\mathsf{Normal}_k/\mathsf{RSS}$ does not. The following example shows this difference between $\mathsf{Normal}_k$ and $\mathsf{Normal}_k/\mathsf{RSS}$.

*Example 1.* Consider a program that has the following call-chain (where $f_1 \overset{c_1,c_2}{\rightarrow} f_2$ denotes that $f_1$ calls $f_2$ at call-sites $c_1$ and $c_2$) and suppose $k = 1$:

$$f_1 \overset{c_1,c_2}{\rightarrow} f_2 \overset{c_3,c_4}{\rightarrow} f_3$$

- $\mathsf{Normal}_1$: The analysis results for $f_2$ are distinguished by $[c_1]$ and $[c_2]$ hence no butterfly cycle happens between $f_1$ and $f_2$. Now, when $f_3$ is called from $f_2$ at $c_3$, we have two call-contexts $(c_3, [c_1])$ and $(c_3, [c_2])$ but analyzing $f_3$ proceeds with context $[c_3]$ (because $k = 1$). That is, $\mathsf{Normal}_k$ forgets the call-context for procedure $f_3$. Thus the result of analyzing $f_3$ must flow back to all call-contexts with return site $c_3$, i.e., to both the call-contexts $(c_3, [c_1])$ and $(c_3, [c_2])$.

– $\mathsf{Normal}_1/\mathsf{RSS}$: The results for $f_2$ and $f_3$ are distinguished in the same way as $\mathsf{Normal}_1$. But, $\mathsf{Normal}_1/\mathsf{RSS}$ additionally remembers the call-contexts for every procedure call. If $f_3$ was called from $c_3$ under context $[c_1]$, our algorithmic technique forces $\mathsf{Normal}_k$ to remember the call-context $(c_3, [c_1])$ for that procedure call. And finishing analyzing $f_3$'s body, $f_3$ returns only to the remembered call-context $(c_3, [c_1])$. This is possible by the one-exclusive-call-per-procedure policy.

We ensure the one-exclusive-call-per-procedure policy by prioritizing a callee over call-sites that (directly or transitively) invoke the callee. The algorithm always analyzes the nodes of the callee $g$ first prior to any other call nodes that invoke $g$: before selecting a work-item as a next job, we exclude from the worklist every call node $call_f^{g,r}$ to $g$ if the worklist contains any node of procedure $h$ that can be reached from $g$ along some call-chain $g \rightarrow \cdots \rightarrow h$, including the case of $g = h$. After excluding such call nodes, the algorithm chooses a work-item in the same way as a normal worklist algorithm.

*Example 2.* Consider a worklist $\{(call_f^{g,r_1},\delta_1),(call_g^{h,r_2},\delta_2),(n_h,\delta_3),(call_h^{i,r_4},\delta_4)\}$ and assume there is a path $f \rightarrow g \rightarrow h$ in the call graph. When choosing a work-item from the worklist, our algorithm first excludes all the call nodes that invoke procedures now being analyzed: $call_g^{h,r_2}$ is excluded because $h$'s node $n_h$ is in the worklist. Similarly, $call_f^{g,r_1}$ is excluded because there is a call-chain $g \rightarrow h$ in the call graph and $h$'s node $n_h$ exists. So, the algorithm chooses a work-item from $\{(n_h,\delta_3),(call_h^{i,r_4},\delta_4)\}$. The excluded work-items $(call_f^{g,r_1},\delta_1)$ and $(call_g^{h,r_2},\delta_2)$ will not be selected unless there are no nodes of $h$ in the worklist.

Fig 3(b) shows our algorithmic technique that is applied to the normal worklist algorithm of Fig 3(a). To transform $\mathsf{Normal}_k$ into $\mathsf{Normal}_k/\mathsf{RSS}$, only shaded lines are inserted; other parts remain the same. *ReturnSite* is a map to record a single return site information (return node and context pair) per procedure. Lines 15-16 are for remembering a single return when encountering a call-site. The algorithm checks if the current node is a call-node and its target procedure is non-recursive (the $\mathsf{recursive}$ predicate decides whether the procedure is recursive or not), and if so, it remembers its single return-site information for the callee. Lines 17-22 handle procedure returns. If the current node is an exit of a non-recursive procedure, only the remembered return for that procedure is used as a next work-item, instead of all possible next (successor, context) pairs (line 23). Prioritizing callee over call nodes is implemented by delaying call nodes to procedures now being analyzed. To do this, in line 12-13, the algorithm excludes the call nodes $\{(call_{\_}^{g,\_},\_) \in \mathcal{W} \mid (n_h,\_) \in \mathcal{W} \wedge \mathsf{reach}(g,h) \wedge \neg\mathsf{recursive}(g)\}$ that invoke non-recursive procedures whose nodes are already contained in the current worklist. $\mathsf{reach}(g,h)$ is true if there is a path in the call graph from $g$ to $h$.

**Correctness and Precision.** One noticeable thing is that the result of our algorithm is not a fixpoint of the given flow equation system, but still a sound approximation of the program semantics. Since the algorithm prunes some computation steps during worklist algorithm (at exit nodes of non-recursive procedures), the result of the algorithm may not be a fixpoint of the original equation

```
(01) : $\delta \in Context = \Delta$
(02) : $w \in Work = Node \times \Delta$
(03) : $\mathcal{W} \in Worklist = 2^{Work}$
(04) : $\mathcal{N} \in Node \times \Delta \to 2^{Node \times \Delta}$
(05) : $State = \Delta \to Mem$
(06) : $\mathcal{T} \in Table = Node \to State$
(07) : $\hat{\mathcal{F}} \in Node \to Mem \to Mem$

(09) : $FixpointIterate\ (\mathcal{W}, \mathcal{T}) =$

(11) : repeat

(13) :     $(n, \delta) := \mathsf{choose}(\mathcal{W})$
(14) :     $m := \hat{\mathcal{F}}\ n\ (\mathcal{T}(n)(\delta))$

(23) :     for all $(n', \delta') \in \mathcal{N}(n, \delta)$ do
(24) :         if $m \not\sqsubseteq \mathcal{T}(n')(\delta')$
(25) :             $\mathcal{W} := \mathcal{W} \cup \{(n', \delta')\}$
(26) :             $\mathcal{T}(n')(\delta') := \mathcal{T}(n')(\delta') \sqcup m$
(27) : until $\mathcal{W} = \emptyset$
```

(a) a normal worklist algorithm $\mathsf{Normal}_k$

```
(01) : $\delta \in Context = \Delta$
(02) : $w \in Work = Node \times \Delta$
(03) : $\mathcal{W} \in Worklist = 2^{Work}$
(04) : $\mathcal{N} \in Node \times \Delta \to 2^{Node \times \Delta}$
(05) : $State = \Delta \to Mem$
(06) : $\mathcal{T} \in Table = Node \to State$
(07) : $\hat{\mathcal{F}} \in Node \to Mem \to Mem$
(08) : $ReturnSite \in ProcName \to Work$

(09) : $FixpointIterate\ (\mathcal{W}, \mathcal{T}) =$
(10) : $ReturnSite := \emptyset$
(11) : repeat
(12) :     $\mathcal{S} := \{(call_{\_}^{g,\_}, \_) \in \mathcal{W} \mid (n_h, \_) \in \mathcal{W} \wedge \mathsf{reach}(g, h) \wedge \neg\mathsf{recursive}(g)\}$
(13) :     $(n, \delta) := \mathsf{choose}(\ \mathcal{W} \setminus \mathcal{S}\ )$
(14) :     $m := \hat{\mathcal{F}}\ n\ (\mathcal{T}(n)(\delta))$
(15) :     if $n = call_f^{g,r} \wedge \neg\mathsf{recursive}(g)$ then
(16) :         $ReturnSite(g) := (r, \delta)$
(17) :     if $n = exit_g \wedge \neg\mathsf{recursive}(g)$ then
(18) :         $(r, \delta_r) := ReturnSite(g)$
(19) :         if $m \not\sqsubseteq \mathcal{T}(r)(\delta_r)$
(20) :             $\mathcal{W} := \mathcal{W} \cup \{(r, \delta_r)\}$
(21) :             $\mathcal{T}(r)(\delta_r) := \mathcal{T}(r)(\delta_r) \sqcup m$
(22) :     else
(23) :         for all $(n', \delta') \in \mathcal{N}(n, \delta)$ do
(24) :             if $m \not\sqsubseteq \mathcal{T}(n')(\delta')$
(25) :                 $\mathcal{W} := \mathcal{W} \cup \{(n', \delta')\}$
(26) :                 $\mathcal{T}(n')(\delta') := \mathcal{T}(n')(\delta') \sqcup m$
(27) : until $\mathcal{W} = \emptyset$
```

(b) our algorithm $\mathsf{Normal}_k/\mathsf{RSS}$

**Fig. 3.** A normal context-sensitive worklist algorithm $\mathsf{Normal}_k$ (left-hand side) and its RSS modification $\mathsf{Normal}_k/\mathsf{RSS}$ (right-hand side). These two algorithms are the same except for shaded regions. For brevity, we omit the usual definition of $\hat{\mathcal{F}}$, which updates the worklist in addition to computing the flow equation's body.

system. However, because the algorithm prunes only spurious returns that definitely do not happen in the real executions of the program, our algorithm does not miss any real executions.

$\mathsf{Normal}_k/\mathsf{RSS}$ is always at least as precise as $\mathsf{Normal}_k$. Because our technique prunes some (worklist-level) computations that occur along invalid return paths, it improves the precision. The actual precision of $\mathsf{Normal}_k/\mathsf{RSS}$ varies depending on the worklist order, but is no worse than that of $\mathsf{Normal}_k$.

*Example 3.* Consider the program in Fig 1 again, and suppose the current worklist is $\{1, 5\}$. When analyzing the program with $\mathsf{Normal}_0$, the fixpoint-solving follows both spurious return paths, regardless of the worklist order,

$$1 \to 2 \to 3 \to 6 \tag{1}$$

$$5 \to 2 \to 3 \to 4 \tag{2}$$

because of multiple returns from node 3. When analyzing with $\mathsf{Normal}_0/\mathsf{RSS}$, there are two possibilities, depending on the worklist order:

1. When $\mathsf{Normal}_0/\mathsf{RSS}$ selects node 1 first: Then the fixpoint iteration sequence may be 1; 2; 3; 4; 5; 2; 3; 6. This sequence involves the spurious path (1)

(because the second visit to node 2 uses the information from node 1 as well as from node 5), but not (2). $\mathsf{Normal}_0/\mathsf{RSS}$ is more precise than $\mathsf{Normal}_0$.

2. When $\mathsf{Normal}_0/\mathsf{RSS}$ selects node 5 first: Then the fixpoint iteration sequence may be $5; 2; 3; 6; 1; 2; 3; 4; 5; 2; 3; 6$. This computation involves both spurious paths (1) and (2). With this iteration order, $\mathsf{Normal}_0$ and $\mathsf{Normal}_0/\mathsf{RSS}$ have the same precision.

## 4 Experiments

We implemented our algorithm inside a realistic C analyzer [5,6]. Experiments with open-source programs show that $\mathsf{Normal}_k/\mathsf{RSS}$ for any $k$ is very likely faster than $\mathsf{Normal}_k$, and that even $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be faster than $\mathsf{Normal}_k$.

### 4.1 Setting Up

$\mathsf{Normal}_k$ is our underlying worklist algorithm, on top of which our industrialized static analyzer [5,6] for C is installed. The analyzer is an interval-domain-based abstract interpreter. The analyzer performs by default flow-sensitive and call-string-based context-sensitive global analysis on the supergraph of the input programs: it computes $\mathcal{T} = \mathit{Node} \to \mathit{State}$ where $\mathit{State} = \Delta \to \mathit{Mem}$. $\mathit{Mem}$ denotes abstract memory $\mathit{Mem} = \mathit{Addr} \to \mathit{Val}$ where $\mathit{Addr}$ denotes abstract locations that are either program variables or allocation sites, and $\mathit{Val}$ denotes abstract values including $\hat{\mathbb{Z}}$ (interval domain), $2^{\mathit{Addr}}$ (addresses), and $2^{\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}}$ (array block, consisting of base address, offset, and size [6]).

We measured the net effects of avoiding spurious interprocedural cycles. Since our algorithmic technique changes the existing worklist order, performance differences between $\mathsf{Normal}_k$ and $\mathsf{Normal}_k/\mathsf{RSS}$ could be attributed not only to avoiding spurious cycles but also to the changed worklist order. In order to measure the net effects of avoiding spurious cycles, we applied the same worklist order to both $\mathsf{Normal}_k$ and $\mathsf{Normal}_k/\mathsf{RSS}$. The order (between nodes) that we used is a reverse topological order between procedures on the call graph: a node $n$ of a procedure $f$ precedes a node $m$ of a procedure $g$ if $f$ precedes $g$ in the reverse topological order in the call graph. If $f$ and $g$ are the same procedure, the order between the nodes are defined by the weak topological order [3] on the control flow graph of the procedure. Note that this ordering itself contains the "prioritize callees over call-sites" feature and we don't explicitly need the delaying call technique (lines 12-13 in Fig 3.(b)) in $\mathsf{Normal}_k/\mathsf{RSS}$. Hence the worklist order for $\mathsf{Normal}_k$ and $\mathsf{Normal}_k/\mathsf{RSS}$ are the same.[4]

We have analyzed 11 open-source software packages. Table 2 shows our benchmark programs as well as their raw analysis results. All experiments were done on a Linux 2.6 system running on a Pentium4 3.2 GHz box with 4 GB of main memory.

[4] In fact, the order described here is the one our analyzer uses by default, which consistently shows better performance than naive worklist management scheme (BFS/DFS) or simple "wait-at-join" techniques (e.g., [6]).

**Table 2.** Benchmark programs and their raw analysis results. Lines of code (**LOC**) are given before preprocessing. The number of nodes in the supergraph(**#nodes**) is given after preprocessing. **k** denotes the size of call-strings used for the analysis. Entries with $\infty$ means missing data because of our analysis running out of memory.

| **Program** | **LOC** | **#nodes** | **k-call-strings** | *#iterations* | | *time* | |
|---|---|---|---|---|---|---|---|
| | | | | Normal | Normal/RSS | Normal | Normal/RSS |
| spell-1.0 | 2,213 | 782 | 0 | 33,864 | 5,800 | 60.98 | 8.49 |
| | | | 1 | 31,933 | 10,109 | 55.02 | 13.35 |
| | | | 2 | 57,083 | 15,226 | 102.28 | 19.04 |
| barcode-0.96 | 4,460 | 2,634 | 0 | 22,040 | 19,556 | 93.22 | 84.44 |
| | | | 1 | 33,808 | 30,311 | 144.37 | 134.57 |
| | | | 2 | 40,176 | 36,058 | 183.49 | 169.08 |
| httptunnel-3.3 | 6,174 | 2,757 | 0 | 442,159 | 48,292 | 2020.10 | 191.53 |
| | | | 1 | 267,291 | 116,666 | 1525.26 | 502.59 |
| | | | 2 | 609,623 | 251,575 | 5983.27 | 1234.75 |
| gzip-1.2.4a | 7,327 | 6,271 | 0 | 653,063 | 88,359 | 4601.23 | 621.52 |
| | | | 1 | 991,135 | 165,892 | 10281.94 | 1217.58 |
| | | | 2 | 1,174,632 | 150,391 | 18263.58 | 1116.25 |
| jwhois-3.0.1 | 9,344 | 5,147 | 0 | 417,529 | 134,389 | 4284.21 | 1273.49 |
| | | | 1 | 272,377 | 138,077 | 2445.56 | 1222.07 |
| | | | 2 | 594,090 | 180,080 | 8448.36 | 1631.07 |
| parser | 10,900 | 9,298 | 0 | 3,452,248 | 230,309 | 61316.91 | 3270.40 |
| | | | 1 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| bc-1.06 | 13,093 | 4,924 | 0 | 1,964,396 | 412,549 | 23515.27 | 3644.13 |
| | | | 1 | 3,038,986 | 1,477,120 | 44859.16 | 12557.88 |
| | | | 2 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| less-290 | 18,449 | 7,754 | 0 | 3,149,284 | 1,420,432 | 46274.67 | 20196.69 |
| | | | 1 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| twolf | 19,700 | 14,610 | 0 | 3,028,814 | 139,082 | 33293.96 | 1395.32 |
| | | | 1 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| tar-1.13 | 20,258 | 10,800 | 0 | 4,748,749 | 700,474 | 75013.88 | 9973.40 |
| | | | 1 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| make-3.76.1 | 27,304 | 11,061 | 0 | 4,613,382 | 2,511,582 | 88221.06 | 44853.49 |
| | | | 1 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

### 4.2 Results

We use two performance measures: (1) *#iterations* is the total number of iterations during the worklist algorithm. The number directly indicates the amount of computation; (2) *time* is the CPU time spent during the analysis.

Fig 4(a) compares the analysis time between $\mathsf{Normal}_k/\mathsf{RSS}$ and $\mathsf{Normal}_k$ for $k = 0, 1, 2$. In this comparison, $\mathsf{Normal}_k/\mathsf{RSS}$ reduces the analysis time of $\mathsf{Normal}_k$ by 7%-96%.

- When $k = 0$ (context-insensitive) : $\mathsf{Normal}_0/\mathsf{RSS}$ has reduced the analysis time by, on average, about 74% against $\mathsf{Normal}_0$. For most programs, the analysis time has been reduced by more than 50%. There is one exception: `barcode`. The analysis time has been reduced by 9%. This is because `barcode` has unusual call structures: it does not call a procedure many times, but calls many different procedures one by one. So, the program contains few butterfly cycles.

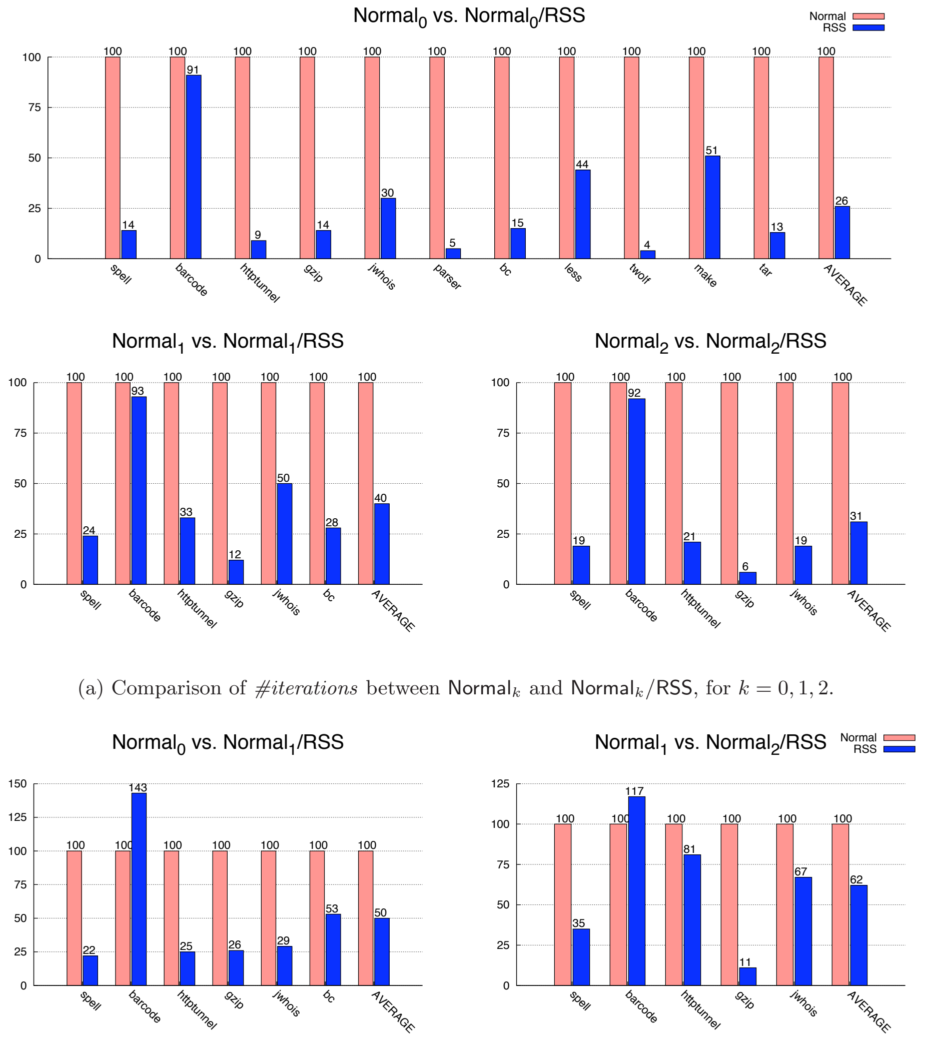
- When $k = 1$: $\mathsf{Normal}_1/\mathsf{RSS}$ has reduced the analysis time by, on average, about 60% against $\mathsf{Normal}_1$. Compared to the context-insensitive case, for all programs, cost reduction ratios have been slightly decreased. This is mainly because, in our analysis, $\mathsf{Normal}_0$ costs more than $\mathsf{Normal}_1$ for most programs (`spell`, `httptunnel`, `jwhois`). For `httptunnel`, in Table 2, the analysis time (2020.10 s) for $k = 1$ is less than the time (1525.26 s) for $k = 0$. This means that performance problems by butterfly cycles is much more severe when $k = 0$ than that of $k = 1$, because by increasing context-sensitivity some spurious paths can be removed.
- When $k = 2$: $\mathsf{Normal}_2/\mathsf{RSS}$ has reduced the analysis time by, on average, 69% against $\mathsf{Normal}_2$. Compared to the case of $k = 1$, the cost reduction ratio has been slightly increased for most programs. In the analysis of $\mathsf{Normal}_2$, since the equation system is much larger than that of $\mathsf{Normal}_1$, our conjecture is that the size of butterfly cycles is likely to get larger. Since larger butterfly cycles causes more serious problems (Section 2), our RSS algorithm is likely to greater reduce useless computation.

Fig 4(b) compares the performance of $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ against $\mathsf{Normal}_k$ for $k = 0, 1$. The result shows that, for all programs except `barcode`, $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ is likely faster than $\mathsf{Normal}_k$. Since $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be even faster than $\mathsf{Normal}_k$, if memory cost permits, we can consider using $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ instead of $\mathsf{Normal}_k$.

Table 3 compares the precision between $\mathsf{Normal}_0$ and $\mathsf{Normal}_0/\mathsf{RSS}$. In order to measure the increased precision, we first joined all the memories associated with each program point (*Node*). Then we counted the number of constant intervals (*#const*, e.g., $[1, 1]$), finite intervals (*#finite*, e.g., $[1, 5]$), intervals with one infinity (*#open*, e.g., $[-1, +\infty)$ or $(-\infty, 1]$), and intervals with two infinity (*#top*, $(-\infty, +\infty)$) from interval values ($\hat{\mathbb{Z}}$) and array blocks ($2^{AllocSite \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}}$) contained in the joined memory. The constant interval and top interval indicate the most precise and imprecise values, respectively. The results show that $\mathsf{Normal}_0/\mathsf{RSS}$ is more precise (`spell`, `barcode`, `httptunnel`, `gzip`) than $\mathsf{Normal}_0$ or the precision is the same (`jwhois`).

**Table 3.** Comparison of precision between $\mathsf{Normal}_0$ and $\mathsf{Normal}_0/\mathsf{RSS}$

| **Program** | **Analysis** | *#const* | *#finite* | *#open* | *#top* |
|---|---|---:|---:|---:|---:|
| spell-1.0 | $\mathsf{Normal}_0$ | 345 | 88 | 33 | 143 |
| | $\mathsf{Normal}_0/\mathsf{RSS}$ | 345 | 89 | 35 | 140 |
| barcode-0.96 | $\mathsf{Normal}_0$ | 2136 | 588 | 240 | 527 |
| | $\mathsf{Normal}_0/\mathsf{RSS}$ | 2136 | 589 | 240 | 526 |
| httptunnel-3.3 | $\mathsf{Normal}_0$ | 1337 | 342 | 120 | 481 |
| | $\mathsf{Normal}_0/\mathsf{RSS}$ | 1345 | 342 | 120 | 473 |
| gzip-1.2.4a | $\mathsf{Normal}_0$ | 1995 | 714 | 255 | 1214 |
| | $\mathsf{Normal}_0/\mathsf{RSS}$ | 1995 | 716 | 255 | 1212 |
| jwhois-3.0.1 | $\mathsf{Normal}_0$ | 2740 | 415 | 961 | 1036 |
| | $\mathsf{Normal}_0/\mathsf{RSS}$ | 2740 | 415 | 961 | 1036 |



(a) Comparison of *#iterations* between $\mathsf{Normal}_k$ and $\mathsf{Normal}_k/\mathsf{RSS}$, for $k = 0, 1, 2$.

(b) Comparison of *#iterations* between $\mathsf{Normal}_k$ and $\mathsf{Normal}_{k+1}/\mathsf{RSS}$, for $k = 0, 1$.

**Fig. 4.** Net effects of avoiding spurious cycles

## 5 Conclusion

We have presented a simple algorithmic technique to alleviate substantial inefficiency in global static analysis caused by large spurious interprocedural cycles. Such cycles are identified as a major reason for the folklore problem in static analysis that less precise analyses sometimes are slower. Although this inefficiency might not come to the fore when analyzing small programs, globally analyzing medium or large programs makes it outstanding. The proposed algorithmic technique reduces the analysis time by 7%-96% for open-source benchmarks.

Though tuning the precision of static analysis can in principle be controlled solely by redesigning the underlying abstract semantics, our algorithmic technique is a simple and orthogonal leverage to effectively shift the analysis cost/ precision balance for the better. The technique's correctness is obvious enough to avoid the burden of a safety proof that would be needed if we newly designed the abstract semantics.

**Acknowledgements.** I am grateful to Wontae Choi, Yungbum Jung, Will Klieber, Soonho Kong, and Daejun Park for their helpful comments and suggestions. I would like to especially thank Deokhwan Kim and Kwangkeun Yi for their very kindhearted help in writing this paper. I am also thankful to anonymous referees for helpful comments.

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF) (R11-2008-007-01002-0) and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009.

## References

1. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 binary executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation, pp. 196–207 (2003)
3. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Proceedings of the International Conference on Formal Methods in Programming and their Applications, pp. 128–141 (1993)
4. Chambers, C., Dean, J., Grove, D.: Frameworks for intra- and interprocedural dataflow analysis. Technical report, Department of Computer Science and Engineering, University of Washington (1996)
5. Jhee, Y., Jin, M., Jung, Y., Kim, D., Kong, S., Lee, H., Oh, H., Park, D., Yi, K.: Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco (January 2008), http://www.ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf
6. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 203–217. Springer, Heidelberg (2005)
7. Karkare, B., Khedker, U.P.: An improved bound for call strings based interprocedural analysis of bit vector frameworks. ACM Trans. on Programming Languages and Systems 29(6), 38 (2007)

8. Khedker, U.P., Karkare, B.: Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 213–228. Springer, Heidelberg (2008)
9. Martin, F.: PAG - an efficient program analyzer generator. International Journal on Software Tools for Technology Transfer 2(1), 46–67 (1998)
10. Martin, F.: Experimental comparison of call string and functional approaches to interprocedural analysis. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 63–75. Springer, Heidelberg (1999)
11. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61 (1995)
12. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. on Programming Languages and System 29(5), 26–51 (2007)
13. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Sicence 167(1-2), 131–170 (1996)
14. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, ch. 7. Prentice-Hall, Englewood Cliffs (1981)
15. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. In: Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation, pp. 387–400 (2006)
16. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation, pp. 131–144 (2004)