ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

ROSAEC MEMO
2009-002
September 16, 2009

Large Spurious Cycle in Global Static Analyses and Its Algorithmic Mitigation

Hakjoo Oh and Kwangkeun Yi

September 16, 2009

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.

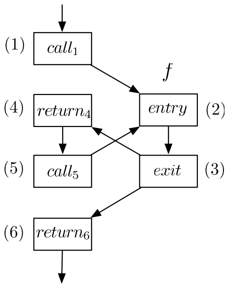
Our simple algorithmic technique (within the fixpoint iteration algorithm) identifies and prunes these spurious interprocedural flows. The technique's effectiveness is proven by experiments with a realistic C analyzer to reduce the analysis time by 7%-96%. Since the technique is *algorithmic*, it can 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 analysis 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,
- such a big spurious cycle makes the conventional call-strings method that distinguishes the last k call-sites [17] 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 part, and
- the algorithmic technique works regardless of the underlying abstract semantics' context-sensitivity (the k). The technique consistently saves the analysis time, without sacrificing (or with even improving) the analysis precision.



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

Figure 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) → $\cdots$

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 Figure 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) → (3) → (4) → (5) → (2) → $\cdots$.

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 the call-strings-based context-sensitive analysis ($k > 0$), large spurious cycles are likely to remain (Section 2).

1.2 Solution: An Algorithmic Mitigation Without Redesigning the Analysis (Abstract Semantics)

We present a simple algorithmic technique inside a worklist-based fixpoint iteration procedure that, without redesigning the abstract semantics part, can effectively relieve the performance degradation caused by spurious interprocedural cycles in both call-strings-based context-sensitive ($k > 0$) and -insensitive ($k = 0$) analysis. For the moment, we consider context-insensitive case only. We extend it to context-sensitive analysis in Section 3.

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 Figure 1) of the "butterfly" cycles. In order to enforce this, we control the equation-solving orders so

---

[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.

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%)
apache-2.2.2	1,364/2,075(66%)	71,719/95,179(75%)

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 Figure 1 again and think of a middle of the analysis (equation-solving) sequence, $\cdots \to (5) \to (2) \to (3)$, 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 $(3) \to (4) \to \cdots$.

We integrated the algorithm inside an industrialized abstract-interpretation-based C static analyzer [6, 7, 8] 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 an 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 [6, 7, 8] in globally analyzing medium-scale open-source programs.

- The technique is meaningful in three ways.

  1. The technique aims to alleviate one major reason (spurious interprocedural cycles) for substantial inefficiency in global static analysis.
  2. 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.
  3. The technique not only reduces the analysis time but also improves the analysis precision. This is because (1) our technique removes some (worklist-level) computations that occur along invalid return paths (Section 3.3.1); (2) when the underlying analysis uses widenings, the technique reduces the number of widening points (Section 3.3.2).

- 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 [12, 14, 16], 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 [17, 1, 11, 12] 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, 11, 12] 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, 7]). 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.

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 [18, 19] 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 [19] or regular-reachability [18] 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 [17, 9, 10] and functional approaches [17] do not suffer from spurious cycles but are limited to restricted classes of data flow analysis problems. The original full call-strings method [17] prescribes the domain to be finite and its improved algorithms [9, 10] are also limited to bit-vector problems or finite domains. Khedker et al.'s algorithm [10] supports infinite domains only for demand-driven analysis. The purely functional approach [17] requires compact representations of flow functions. The iterative (functional) approach [17] requires the domain to be finite.

Reps et al.'s algorithms [13, 15] 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 [5] with non-distributive flow functions does not fall into either their IFDS [13] or IDE [15] problems. Meanwhile, our algorithm is independent of the underlying abstract semantic functions. The regular-reachability [18], which is a restricted version of Reps et al.'s algorithm [13], also requires the analysis problem to be expressed in graph reachability problem.

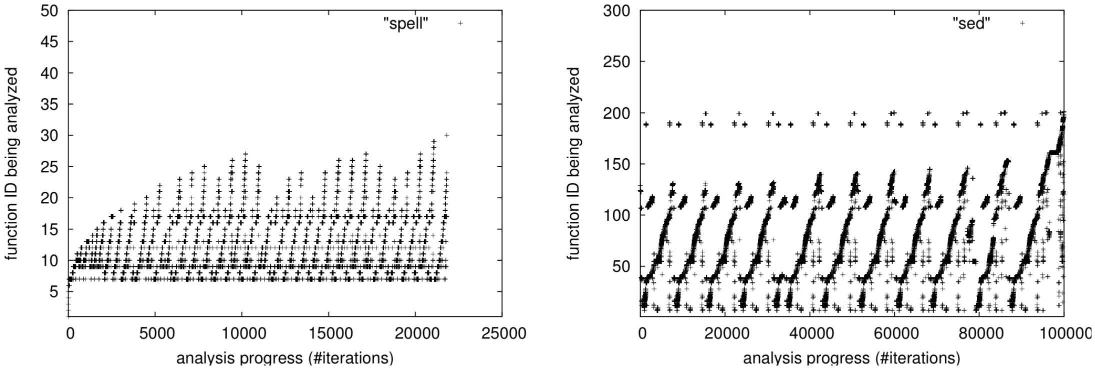
Chambers et al.'s technique [4] is similar to ours but entails a relatively large change to an existing worklist order. Their technique analyzes each procedure intraprocedurally, and at call-sites continues the analysis of the callee. It returns to analyze the nodes of the caller only after finishing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

1.5 Organization

Section 2 discusses the performance problem of the traditional call-strings-based context-sensitive or -insensitive interprocedural analysis. Section 3 presents our solution to mitigate the

---

[3] For example, such analyzers include octagon-based analyzers (e.g.,[2]), interval-based analyzers (e.g.,[6, 7, 8]), value set analysis [1], and program analyzer generators (e.g, [11]), which usually use infinite (height) domains and non-distributive flow functions.



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

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

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

problem. We first describe the classical call-strings approach and then present our extension of the original method. Section 4 presents experimental results that compare the performance of our algorithm with the traditional algorithm. Section 5 concludes the paper.

2 Performance Problems by Large Spurious Cycles

In this section, we show that large spurious cycles are frequently created during (both context-insensitive and -sensitive) global static analysis, and that they drastically degrade the analysis performance. The classical call-strings-based context-sensitive abstract semantics cannot effectively eliminate such large spurious cycles.

2.1 Interprocedural Spurious Cycles Reach Far In Real C Programs

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.

In conventional context-sensitive analysis that distinguishes the last k call-sites [17], 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. Figure 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 present our extension of the classical call-strings-based approach, aiming to mitigate performance problems caused by the large spurious cycles. Our technique is purely algorithmic: the technique does not depend on the underlying abstract semantics but is a simple addition to the existing worklist-based fixpoint algorithm.

We first describe the traditional call-strings-based analysis algorithm (section 3.2) as well as the representation of programs (section 3.1). Then we present our algorithmic extension of the classical algorithm (section 3.3).

3.1 Graph Representation of Programs

We assume that a program is represented by a supergraph [13]. A supergraph consists of control flow graphs of procedures with interprocedural edges connecting each call-site to its callee. Each node $n \in \mathit{Node}$ in the graph has one of the five types :

$$\mathit{entry}_f \mid \mathit{exit}_f \mid \mathit{call}_f^{g,r} \mid \mathit{rtn}_f^c \mid \mathit{cmd}_f$$

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 $\mathit{call}_f^{g,r}$ 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, succof, which maps each node to its successors. *CallNode* is the set of call nodes in a program.

3.2 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 $\mathit{CallNode}^{\leq k} \overset{\text{let}}{=} \Delta$ 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 δ and a call node c, $[\delta, c]$ denotes a call-string obtained by appending c to δ. In the case of context-insensitive analysis ($k = 0$), we use $\Delta = \{\epsilon\}$, where the empty call-string ϵ means no context-information.

Figure 3.(a) shows the worklist-based fixpoint iteration algorithm that performs call-strings(Δ)-based context-sensitive (or insensitive, when $k = 0$) analysis. The algorithm computes a table $\mathcal{T} \in \mathit{Node} \to \mathit{State}$ which associates each node with its input state $\mathit{State} = \Delta \to \mathit{Mem}$, 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 $(n, \delta) \in \mathit{Node} \times \Delta$ 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 function $\mathcal{N} \in \mathit{Node} \times \Delta \to 2^{\mathit{Node} \times \Delta}$:

$$\mathcal{N}(n, \delta) = \begin{cases} \{(r, \delta') \mid \delta = \lceil \delta', \mathit{call}_f^{g,r} \rceil_k \wedge \delta' \in \mathsf{dom}(\mathcal{T}(\mathit{call}_f^{g,r}))\} & \text{if } n = \mathit{exit}_g \\ \{(\mathit{entry}_g, \lceil \delta, n \rceil_k))\} & \text{if } n = \mathit{call}_f^{g,r} \\ \{(n', \delta) \mid n' \in \mathsf{succof}(n)\} & \text{otherwise} \end{cases}$$

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 δ' 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. The following example illustrates this situation.

Example 1 *Let* $k = 2$ *and suppose call-strings* $[c_1, c_3]$ *and* $[c_2, c_3]$ *are tagged to a call node* $call_f^{g,r}$*. Suppose* $call_f^{g,r}$ *calls* g *under the call-string* $[c_1, c_3]$*. By the definition of* $\mathcal{N}$*, the call-string at* $entry_g$ *is* $\lceil c_1, c_3, call_f^{g,r}\rceil_2 = [c_3, call_f^{g,r}]$*. After the analysis of* g*, the call-string at* $exit_g$ *is also* $[c_3, call_f^{g,r}]$*. When* g *returns, since the call-string at* $exit_g$ *equals to both* $\lceil c_1, c_3, call_f^{g,r}\rceil_2$ *and* $\lceil c_2, c_3, call_f^{g,r}\rceil_2$*,* $\mathcal{N}$ *returns two work-items* $(r, [c_1, c_3])$ *and* $(r, [c_2, c_3])$*. The return to* $(r, [c_2, c_3])$ *is spurious because* g *was called under the context* $[c_1, c_3]$*.* □

We call the above analysis algorithm $\mathsf{Normal}_k (k = 0, 1, 2, \dots)$. Normal_0 performs context-insensitive analysis, Normal_1 performs context-sensitive analysis that distinguishes the last 1 call-site, and so on.

3.3 Normal_k/RSS: Our Algorithm

Before discussing our technique, we define the call-context that will be used throughout this section.

Definition 1 *When a procedure* g *is called from a call node* $call_f^{g,r}$ *under context* δ*, 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, δ) *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 δ, the algorithm remembers its call-context (r, δ), consisting of the corresponding return node r and the call-string δ. And upon finishing analyzing g's body, after evaluating $exit_g$, the algorithm inserts only the remembered return node and its call-string (r, δ) into the worklist. Multiple returns are avoided. For correctness, this single return should be allowed only when the other call nodes that call g are not analyzed until the analysis of g from $(call_f^{g,r}, \delta)$ completes.

   Example 2 *Consider the situation of Example 1 again. When* g *is called from* $call_f^{g,r}$ *under the context* $[c_1, c_3]$*, our algorithm remembers* g*'s call-context* $(r, [c_1, c_3])$*. And at* $exit_g$*, under its context* $[c_3, call_f^{g,r}]$*, our algorithm inserts only the remembered* $(r, [c_1, c_3])$ *into the worklist. The spurious return to* $(r, [c_2, c_3])$ *is avoided.* □

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 δ, processings of other call-sites that call the same procedure should wait until the analysis of the procedure from (c, δ) is completely finished. This one-exclusive-call-per-procedure policy is enforced by not selecting from the worklist call nodes that (directly or transitively) call the procedures that are currently being analyzed.

Example 3 *Suppose procedure g was called from $call_f^{g,r}$ under the context $[c_1, c_3]$ and our algorithm has remembered the call-context $(r, [c_1, c_3])$. Suppose also the current worklist $\mathcal{W} = \{(call_f^{g,r}, [c_2, c_3]), \cdots\}$ which contains a call-site that invokes g. In this situation, our algorithm does not select $(call_f^{g,r}, [c_2, c_3])$ as a next work-item unless the analysis of g is completely finished.*

□

3. **Recursion handling**: The algorithm gives up the single return policy for recursive procedures. This is because we cannot finish analyzing a recursive procedure's body without considering another call (recursive call) 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 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 Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$.

Example 4 *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 $k = 1$:*

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

- *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, 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 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 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, i.e., after the exclusion, our algorithm relies on the existing worklist ordering strategy in selecting the next work-item.

Example 5 *Consider a worklist $\{(call_f^{g,r_1}, \delta_1), (call_j^{h,r_2}, \delta_2), (n_h, \delta_3), (call_h^{i,r_4}, \delta_4)\}$ 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: $call_j^{h,r_2}$ 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 $(call_j^{h,r_2}, \delta_2)$ will not be selected unless there are no nodes of h in the worklist.* □

```
(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 Normal_k

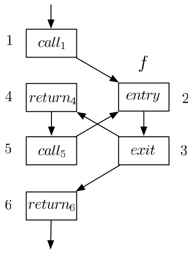
```
(01) : $\delta \in Context = \Delta$
(02) : $w \in Work = Node \times \Delta$
(03) : $\mathcal{W} \in Worklist = 2^{Work}$
(04) : $\mathcal{N} \in Node \times \Delta \to 2^{Node \times \Delta}$
(05) : $State = \Delta \to Mem$
(06) : $\mathcal{T} \in Table = Node \to State$
(07) : $\hat{\mathcal{F}} \in Node \to Mem \to Mem$
(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 $n = call^{g,r}_{f} \wedge \neg\mathsf{recursive}(g)$ 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 $\mathsf{Normal}_k/\mathsf{RSS}$

Figure 3: A normal context-sensitive worklist algorithm Normal_k and its RSS modification $\mathsf{Normal}_k/\mathsf{RSS}$. The left-hand side shows a worklist algorithm for call-strings based context-sensitive analysis. The right-hand side shows the RSS algorithm for the same analysis. 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.

Figure 3(b) shows our algorithmic technique that is applied to the normal worklist algorithm of Figure 3(a). To transform 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 17-21 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.

Example 6 *Analyzing the program in the left-hand side of Figure 4 proceeds as shown in the right-hand side table. (Assume that $k = 0$, the* choose *function in Figure 3 arbitrarily chooses an element from the given worklist, and the initial worklist is $\{1, 4\}$). For each iteration of*



Iters	$\mathcal{W}$	$\mathcal{S}$	$\mathcal{W} \setminus \mathcal{S}$	*ReturnSite*	updated $\mathcal{W}$
1	$\{1,4\}$	$\{\}$	$\{\bar{1}\}$	$\{f \mapsto 4\}$	$\{2,4\}$
2	$\{2,4\}$	$\{\}$	$\{2,\bar{4}\}$	$\{f \mapsto 4\}$	$\{2,5\}$
3	$\{2,5\}$	$\{5\}$	$\{\bar{2}\}$	$\{f \mapsto 4\}$	$\{3,5\}$
4	$\{3,5\}$	$\{5\}$	$\{\bar{3}\}$	$\{f \mapsto 4\}$	$\{4,5\}$

Figure 4: A running example of our algorithm (Figure 3).

*the algorithm, the table shows the contents of the current worklist ($\mathcal{W}$), call nodes that are excluded at this iteration ($\mathcal{S}$), return site information (*ReturnSite*), and the updated worklist ($\mathcal{W}$). $\bar{n}$ represents the chosen node for each iteration. When the algorithm processes call node* 1 *at the first iteration, f remembers its corresponding return-site* 4*. At the 3rd and 4th iterations, node* 5 *was excluded, because it is another call to f and the worklist contains the nodes of f at both iterations. At the exit of f (when processing node* 3 *at the 4th iteration), only* $\mathrm{ReturnSite}(f) = 4$ *is inserted into the worklist instead of* $\mathsf{succof}(f) = \{4, 6\}$. ☐

3.3.1 Correctness & Precision

One noticeable thing of $\mathsf{Normal}_k/\mathsf{RSS}$ is that the result 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 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 information flow of real executions.

For any f and any arbitrary call-context $(call_g^{f,r}, \delta)$, the single return to (r, δ) after analyzing f is correct if the state from $(call_g^{f,r}, \delta)$ is implied by the input state used in the analysis of f and its result is guaranteed to be returned to (r, δ). The state from every call-context flows into f (abstract semantics). Our single-return policy does not miss returning f's analysis result to its corresponding call-context[4] because (1) we remember the context at each call; (2) for every different call, modulo the underlying context-sensitivity, we exclusively analyze f. Because we cannot enforce this exclusivity for recursive calls, we do not apply the algorithm to recursive procedures.

$\mathsf{Normal}_k/\mathsf{RSS}$ is always at least as precise as Normal_k. Because $\mathsf{Normal}_k/\mathsf{RSS}$ prunes some (worklist-level) computations that occur along invalid return paths, it is likely to have an effect of avoiding propagations of information along invalid return paths. Hence, $\mathsf{Normal}_k/\mathsf{RSS}$ gives more precise (or at least the same) results than Normal_k. The actual precision of $\mathsf{Normal}_k/\mathsf{RSS}$ varies depending on the existing worklist order of Normal_k.

Example 7 *Consider the program in Figure 4 again, and suppose the current worklist is* $\{1, 5\}$. *When analyzing the program with* Normal_0, *the fixpoint-solving follows both spurious return paths, regardless of the worklist order,*

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

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

---

[4] Here, we ignore the cases where the callee never returns (e.g., it calls exit()). However, even though that happens, we can enforce the return of callee by always inserting the exit node of a procedure when inserting the entry node of the procedure into the worklist.

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

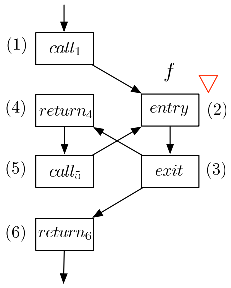
1. *When* $\mathsf{Normal}_0/\mathsf{RSS}$ *selects node 1 first: Then the fixpoint iteration sequence may be* $1; 2; 3; 4; 5; 2; 3; 6$*. This sequence involves the spurious path (1) (because the second visit to node 2 uses the information from node 1 as well as from node 5), but not (2).* $\mathsf{Normal}_0/\mathsf{RSS}$ *is more precise than* 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,* Normal_0 *and* $\mathsf{Normal}_0/\mathsf{RSS}$ *have the same precision.*

□

3.3.2 Less Widening Points

Our technique reduces cycles, hence obviously reduces the number of widening points. For analyses with infinite or very large height domains such as lattice of intervals, the widening technique [5] is used to guarantee or accelerate the analysis' termination. Because applying widening means losing analysis precision, the widening operation should be carefully applied to as small as possible subset of the entire program points. A common way of selecting such widening points is to apply widening to every heads of loops in program [3], including ones that are interprocedurally created by calling a procedure multiple times. $\mathsf{Normal}_k/\mathsf{RSS}$ can reduce the number of widening points more. $\mathsf{Normal}_k/\mathsf{RSS}$ need not apply widenings at interproceudral loop-heads that are created by non-recursive procedure calls. This is because $\mathsf{Normal}_k/\mathsf{RSS}$ does not follow such interprocedural cycles. For example, consider the following code and interval-domain-based analysis of the code.

```
int g = 0;
int f() { g++; }
int main() {
   f();
   f();
}
```



Since procedure `f` is called twice from procedure `main`, a spurious interprocedural cycle $(5) \to (2) \to (3) \to (4) \to (5) \cdots$ will be created during the analysis. Iterating through the cycle continually increases the value of the global variable `g`: $[0, 0] \to [0, 1] \to [0, 2] \to \cdots$. In order to terminate the analysis, a widening should be applied at the entry of procedure `f`. Hence, Normal_k computes $\mathtt{g} = [0, +\infty]$ at the end of procedure `main`. However, $\mathsf{Normal}_k/\mathsf{RSS}$ does not apply the widening at the entry of procedure `f` (since `f` is non-recurisve and $\mathsf{Normal}_k/\mathsf{RSS}$ does not follow the spurious return paths $(5) \to (2) \to (3) \to (4)$), computing $\mathtt{g} = [0, 2]$ at the end of procedure `main`.

4 Experiments

We implemented our algorithm inside a realistic C analyzer [6, 7, 8]. Experiments with open-source programs show that $\mathsf{Normal}_k/\mathsf{RSS}$ for any k is very likely faster than Normal_k, and that even $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be faster than Normal_k.

Table 2: Benchmark programs and their raw analysis results when using RevTop worklist order. 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 ∞ 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	∞	∞	∞	∞
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	∞	∞	∞	∞
less-290	18,449	7,754	0	3,149,284	1,420,432	46274.67	20196.69
			1	∞	∞	∞	∞
twolf	19,700	14,610	0	3,028,814	139,082	33293.96	1395.32
			1	∞	∞	∞	∞
tar-1.13	20,258	10,800	0	4,748,749	700,474	75013.88	9973.40
			1	∞	∞	∞	∞
make-3.76.1	27,304	11,061	0	4,613,382	2,511,582	88221.06	44853.49
			1	∞	∞	∞	∞

4.1 Setting Up

Normal$_k$ is our underlying worklist algorithm, on top of which our industrialized static analyzer [6, 7, 8] for C is installed. The analyzer is an interval-domain-based abstract interpreter. The analyzer performs by default flow-sensitive and call-strings-based context-sensitive global analysis on the supergraph of the input program: it computes $\mathcal{T} = \mathit{Node} \rightarrow \mathit{State}$ where $\mathit{State} = \Delta \rightarrow \mathit{Mem}$. Mem denotes abstract memory $\mathit{Mem} = \mathit{Addr} \rightarrow \mathit{Val}$ where Addr denotes abstract locations that are either program variables or allocation sites, and Val denotes abstract values including $\hat{\mathbb{Z}}$ (interval domain), 2^{Addr} (points-to set), and $2^{\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}}$ (array block, consisting of base address, offset, and size [7]).

We evaluated our algorithm in two ways. First, we measured the net effects of avoiding spurious interprocedural cycles. Since our algorithmic technique changes the existing worklist order, performance differences between Normal$_k$ and Normal$_k$/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 Normal$_k$ and Normal$_k$/RSS. To be specific, 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. We call the order RevTop order. 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 Figure 3.(b)) in Normal$_k$/RSS. Hence the worklist order for Normal$_k$ and Normal$_k$/RSS are the same.[5] For this evaluation, we compare

---

[5] In fact, the order described here is the one our analyzer uses by default, which consistently shows better

analysis time and precision between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$.

We also evaluated our algorithm when our technique interferes with the existing worklist order. Because our technique interferes with (i.e., changes) the existing worklist order of Normal_k, it is necessary to check whether our technique works well regardless of the existing worklist order strategies or not. To see what happens in this case, we applied our technique to Normal_k that uses the following worklist order, called Arbitrary; the order between nodes in different procedures is determined by a random order that is fixed before the analysis and the order between nodes in the same procedure is defined by the weak topological order. Note that the worklist order does not contain the "prioritize callees over call-sites" because the order randomly chooses a procedure regardless of call relationship.

We have analyzed 11 open-source and SPEC2000 software packages. *Table* 2 shows our benchmark programs. All experiments were done on a Linux 2.6 system running on a Pentium4 3.2 GHz box with 4 GB of main memory. `parser` and `twolf` are from SPEC2000 benchmarks and the others are open-source software.

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. Though *time* is roughly proportional to *#iterations*, it is subject to change because of different implementations and test environments.

4.2 The Net Effects of Avoiding Spurious Cycles

4.2.1 Reduced Analysis Time

Figure 5.(a) compares *#iterations* between $\mathsf{Normal}_k/\mathsf{RSS}$ and Normal_k for $k = 0, 1, 2$ using RevTop worklist order, which shows the net effects of avoiding spurious cycles. In this comparison, $\mathsf{Normal}_k/\mathsf{RSS}$ reduces the number of iterations of Normal_k by on average 72%.

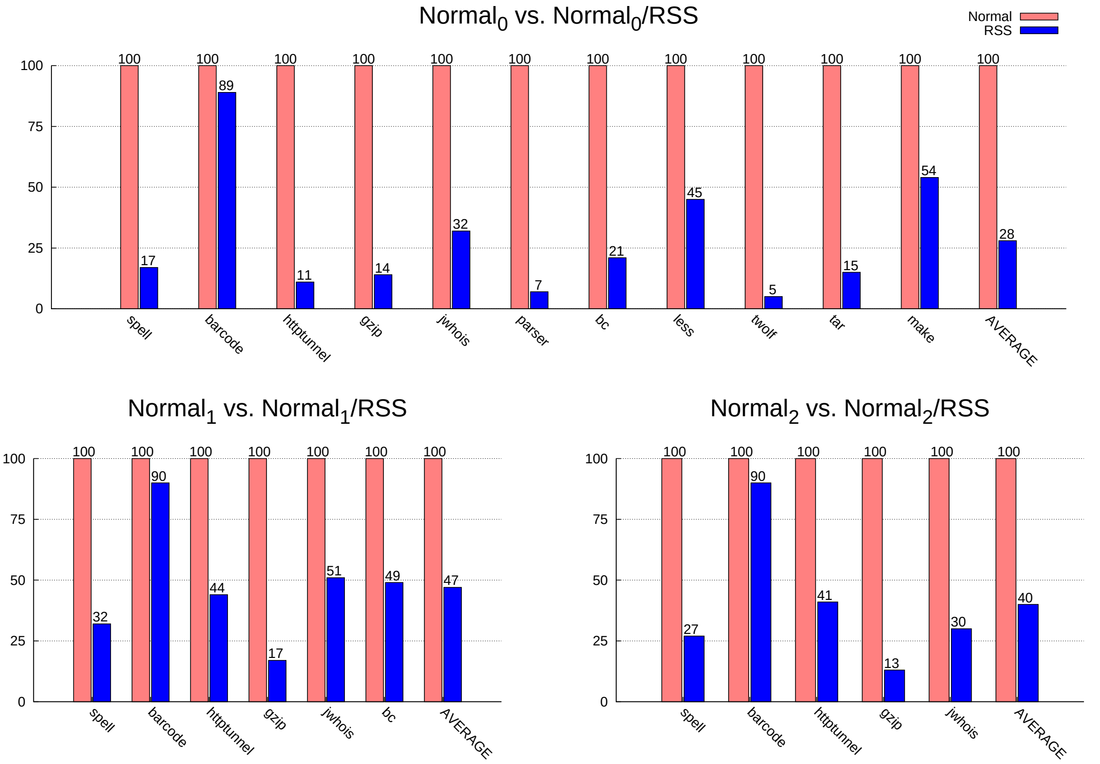
- When $k = 0$ (context-insensitive) : $\mathsf{Normal}_0/\mathsf{RSS}$ has reduced *#iterations* by, on average, about 72% against Normal_0. For most programs, the analysis time has been reduced by more than 50%. There is one exception: `barcode`. The amount of computation has been reduced by 11%. This 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 *#iterations* by, on average, about 53% against Normal_1. Compared to the context-insensitive case ($k = 0$), for all programs, cost reduction ratios have been slightly decreased. As an example, for `spell`, the reduction ratio when $k = 0$ is 83% and the ratio when $k = 1$ is 68%. This is mainly because, in our analysis, Normal_0 costs more than 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. However, by using our algorithm, we can still reduce the cost of Normal_1 by 53%.

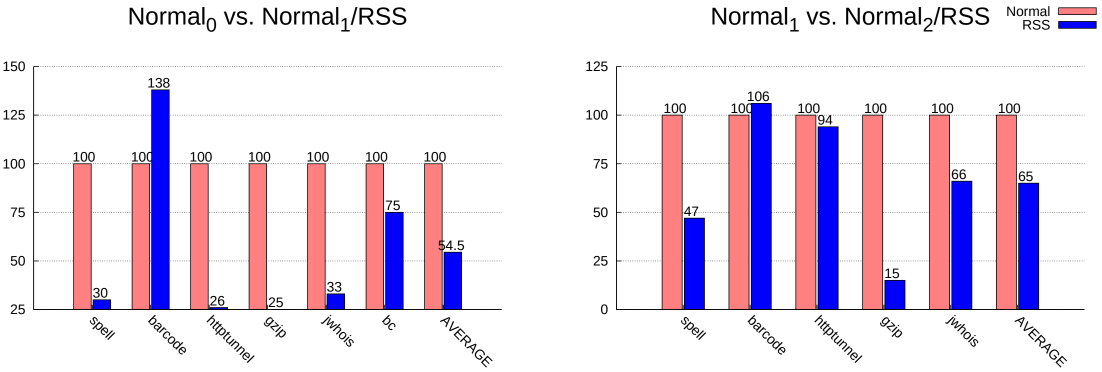
- When $k = 2$: $\mathsf{Normal}_2/\mathsf{RSS}$ has reduced *#iterations* by, on average, 60% against Normal_2. Compared to the case of $k = 1$, the cost reduction ratio has been slightly increased for most programs. For example, the ratio for `spell` has changed from 68% to 73%. In the analysis of Normal_2, since the equation system is much larger than that of 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.

---

performance than naive worklist management scheme (BFS/DFS) or simple "wait-at-join" techniques (e.g., [7]).



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



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

Figure 5: Net effects of avoiding spurious cycles

Table 3: Comparison of precision between Normal_0 and $\mathsf{Normal}_0/\mathsf{RSS}$.

Program	**Analysis**	*#const*	*#finite*	*#open*	*#top*
spell-1.0	Normal_0	345	88	33	143
	$\mathsf{Normal}_0/\mathsf{RSS}$	345	89	35	140
barcode-0.96	Normal_0	2136	588	240	527
	$\mathsf{Normal}_0/\mathsf{RSS}$	2136	589	240	526
httptunnel-3.3	Normal_0	1337	342	120	481
	$\mathsf{Normal}_0/\mathsf{RSS}$	1345	342	120	473
gzip-1.2.4a	Normal_0	1995	714	255	1214
	$\mathsf{Normal}_0/\mathsf{RSS}$	1995	716	255	1212
jwhois-3.0.1	Normal_0	2740	415	961	1036
	$\mathsf{Normal}_0/\mathsf{RSS}$	2740	415	961	1036

Figure 5.(b) compares the performance of $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ against Normal_k for $k = 0, 1$. The result shows that, for all programs except `barcode`, even $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ is faster than Normal_k. Since $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be even faster than Normal_k, if memory cost permits, we can consider using $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ instead of Normal_k.

4.2.2 Increased Analysis Precision

Table 3 compares the precision between Normal_0 and $\mathsf{Normal}_0/\mathsf{RSS}$.[6] 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 Normal_0 or the precision is the same (`jwhois`).

4.3 Speed Up When Interfering the Existing Worklist Order

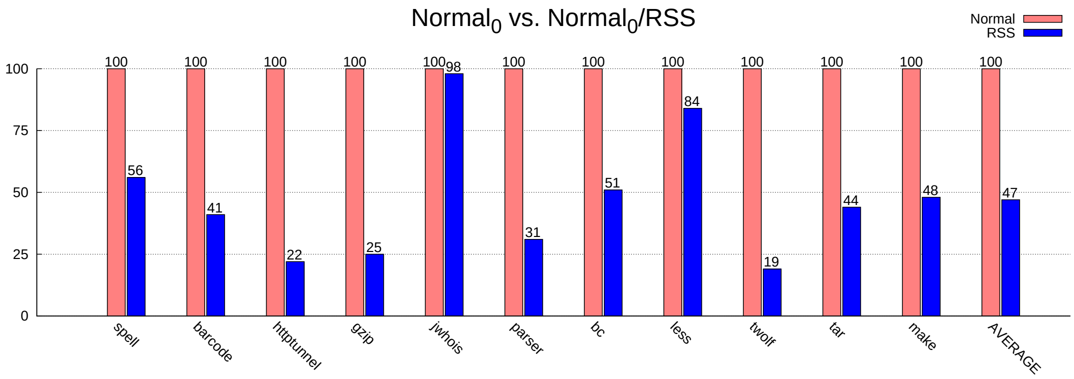
Figure 6.(a) compares *#iterations* between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$ for $k = 0$ using Arbitrary worklist order. In the comparison, $\mathsf{Normal}_k/\mathsf{RSS}$ reduces the computation cost of Normal_k by on average 53%. From this results, we can find that the interference does not significantly affect the overall performance differences: the reduction ratio has been decreased by 19% from the case of net effects of avoiding spurious cycles (72%). Hence, the technique is likely to relieve the problems of spurious cycles regardless of the existing worklist ordering strategies.

5 Conclusion

We have presented a simple algorithmic extension of the classical call-strings approach to alleviate substantial inefficiency 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.

---

[6] We compared the precision for the case of $k = 0$ and for the first five programs in *Table* 2 because we need more memory to do the precision comparison (we should keep two analysis results of Normal_0 and $\mathsf{Normal}_0/\mathsf{RSS}$ at the same time).



(a) Comparison of of *#iterations* between Normal$_0$ and Normal$_0$/RSS, for $k = 0$.

Program	*#iterations*		*time*	
	Normal	Normal/RSS	Normal	Normal/RSS
spell-1.0	36,272	20,377	99.19	43.66
barcode-0.96	71,342	29,574	534.9	154.36
httptunnel-3.3	591,030	132,668	4132.21	730.95
gzip-1.2.4a	804,240	204,553	6844.31	1299.36
jwhois-3.0.1	777,867	761,117	5518.04	4664.2
parser	3,500,035	1,095,194	70248.32	24249.95
bc-1.06	2,231,064	1,138,847	23136.25	14240.14
less-290	3,118,068	2,613,384	53152.72	66329.59
twolf	3,347,610	645,922	52372.78	7179.93
tar-1.13	5,310,745	2,334,886	92637.58	78013.96
make-3.76.1	4,415,305	2,110,272	70553.14	43381.18

(b) Benchmark programs and their raw analysis results.

Figure 6: The analysis results when using Arbitrary worklist order.

Our technique is orthogonally applicable to context-sensitive analysis. It is a simple technique inside the worklist-based fixpoint iteration routine. It is directly applicable without changing the analysis' underlying abstract semantics, regardless of whether the semantics is context-sensitive or not.

Our technique suggests the following implementation guideline in tuning a global semantic analysis. Suppose we develop an analyzer that uses call-strings of size k for context-sensitivity with the Normal_k algorithm. Suppose further that we cannot increase the call-strings size more than k because of either the time or memory cost. In this situation, our algorithmic technique has the following usages.

- When Normal_k hits the memory cost limit: then use $\mathsf{Normal}_k/\mathsf{RSS}$ instead. This is because (1) $\mathsf{Normal}_k/\mathsf{RSS}$ is empirically faster than Normal_k (Section 4.1 and Figure 5.(a),6); (2) $\mathsf{Normal}_k/\mathsf{RSS}$ is in principle more accurate or at least does not sacrifice the precision of Normal_k (Section 3.3.1, 3.3.2 and *Table* 3); (3) $\mathsf{Normal}_k/\mathsf{RSS}$ requires in extra just as many memory entities as the number of procedures.
- When Normal_k hits the time cost limit: then, if memory permits, consider using $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ instead. This is because (1) $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be even faster than Normal_k (Section 4.1 and Figure 5.(b)); (2) it requires in extra just as many entities as the number of procedures.

Though tuning the accuracy 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/accuracy balance for the better. The technique's correctness is obvious enough to avoid the burden of safety proof of otherwise a newly designed abstract semantics.

References

[1] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 binary executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 196–207, 2003.

[3] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993.

[4] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical report, Department of Computer Science and Engineering, University of Washington, 1996.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] Yongin Jhee, Minsik Jin, Yungbum Jung, Deokhwan Kim, Soonho Kong, Heejong Lee, Hakjoo Oh, Daejun Park, and Kwangkeun Yi. Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco, `ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf`, January 2008.

[7] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *Proceedings of the International Symposium on Static Analysis*, pages 203–217, 2005.

[8] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the International Symposium on Memory Management*, pages 131–140, 2008.

[9] Bageshri Karkare and Uday P. Khedker. An improved bound for call strings based interprocedural analysis of bit vector frameworks. *ACM Trans on Programming Languages and Systems*, 29(6):38, 2007.

[10] Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Proceedings of the International Conference on Compiler Construction*, pages 213–228, 2008.

[11] Florian Martin. PAG - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[12] Florian Martin. Experimental comparison of call string and functional approaches to interprocedural analysis. In *Proceedings of the International Conference on Compiler Construction*, pages 63–75, 1999.

[13] Thomas Reps, Susan Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[14] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans on Programming Languages and System*, 29(5):26–51, 2007.

[15] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Sicence*, 167(1-2):131–170, 1996.

[16] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the International Symposium on Static Analysis*, pages 16–34, 1997.

[17] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[18] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[19] John Whaley and Monica S. Lam. 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*, pages 131–144, 2004.