Bottom-up and Top-down Context-Sensitive Summary-Based Pointer Analysis

Erik M. Nystrom, Hong-Seok Kim, Wen-mei W. Hwu

Center for Reliable and High Performance Computing
University of Illinois, Urbana-Champaign

http://www.crhc.uiuc.edu/Impact

Abstract. This paper addresses scalability and accuracy of summary-based context-sensitive pointer analysis formulated as a two-phase computation. The first phase, or *bottom-up* phase, propagates procedure summaries from callees to callers. Then, the second phase, or *top-down* phase, computes the actual pointer information. These two phases can be independently context-sensitive. Having observed the problems that procedural side effects cause, we developed a bottom-up phase that constructs concise procedure summaries in a manner that permits their subsequent removal. This transformation results in an efficient two-phase pointer analysis in the style of Andersen [1] that is simultaneously bottom-up and top-down context-sensitive. Context sensitivity becomes inherent to even a context-insensitive analysis allowing for an accurate and efficient top-down phase. The implemented context-sensitive analysis exhibits scalability comparable to that of its context-insensitive counterpart. For instance, to analyze 176.gcc, the largest C benchmark in SPEC 2000, our analysis takes 190 seconds as opposed to 44 seconds for the context-insensitive analysis. Given the common practice of treating recursive subgraphs context-insensitively, its accuracy is equivalent to an analysis which completely inlines all procedure calls.

1 Introduction

Modern programming practices encourage code reuse, which often results in programs composed of a complex network of procedure calls. For static analysis, this exacerbates the problem of unrealizable (or spurious) interprocedural data flow [24]. *Context-sensitive* analyses are often able to avoid unrealizable data flow through procedure calls thereby delivering a higher degree of accuracy than their context-insensitive counterparts.

In the literature, there have been many approaches to context-sensitive pointer analysis. Some analyses [8, 16, 19, 25, 27] mimic dynamic execution, repeating call-return sequences until the analysis reaches a global fixed point. In these analyses, a procedure must be re-analyzed whenever there is a change in its callers or callees, leading to serious scalability issues. More recent work [2, 3, 12, 20, 21] formulates pointer analysis as a two-phase computation. This formulation

has the advantage that it can be designed to analyze a procedure at most twice when given a fixed acyclic call graph.

In the first phase of the two-phase computation, called the *bottom-up* phase, a flow function is constructed for each procedure by propagating *procedure summaries* from callees to callers. Then, the second phase, called the *top-down* phase, computes the actual pointer information by propagating calling contexts from callers to callees.

Each phase may be independently context-sensitive or -insensitive. On one hand, from the perspective of a caller, distinct calls to the same procedure should be analyzed independently. We call this *bottom-up* context sensitivity. On the other hand, from the perspective of a callee, its calling contexts from distinct call sites should be treated independently. We call this *top-down* context sensitivity.

Previous work [2, 3, 12, 20, 21] primarily focused on the scalability of bottom-up context sensitivity and provided little discussion about scalable top-down context sensitivity. When designed to analyze each procedure at most twice, the top-down phase in previous analyses has the following limitations:

1. The merging of calling contexts from distinct call sites before analyzing a callee leads to spurious interprocedural data flow.
2. The propagation of calling context information from the outputs of callers' flow functions to the inputs of callees' flow functions involves significant copying overheads, impacting scalability.

While investigating these limitations, we realized the importance of *procedural side effects* (loosely, a callee's effects on a caller) in dealing with context sensitivity. If a program lacks procedural side effects, a context-insensitive analysis does not derive any unrealizable interprocedural data flow. Based on this finding, we have reformulated summary-based analysis as follows:

1. The bottom-up phase transforms a program into a form that lacks procedural side effects, yet, has the same overall pointer behavior. This involves the *cutting* and *pasting* of procedural side effects from callees to callers instead of the *copying* and *pasting* performed by previous work [2, 3, 12, 20, 21].
2. The top-down phase consists of a single run of the context-insensitive analysis. Since the program is free of side effects after the bottom-up phase, the context-insensitive analysis does not experience any unrealizable interprocedural data flow. Therefore, in this form, top-down context sensitivity is inherent without any need to copy calling contexts during the top-down propagation.

To demonstrate the effectiveness of the proposed scheme, we implemented a pointer analysis for C programs with the following characteristics.

1. Intraprocedurally, it is a variant of the Andersen's algorithm. In particular, we use the formulation by Heintze and Tardieu [14]. Fields are handled in an offset-based manner while array indices are ignored. For each heap allocation site, we introduce a unique global variable.
2. Procedures in a common call graph cycle are merged into a single procedure. Effectively, recursion is handled in a context-insensitive way. Indirect calls are handled optimistically and iteratively (Section 3).

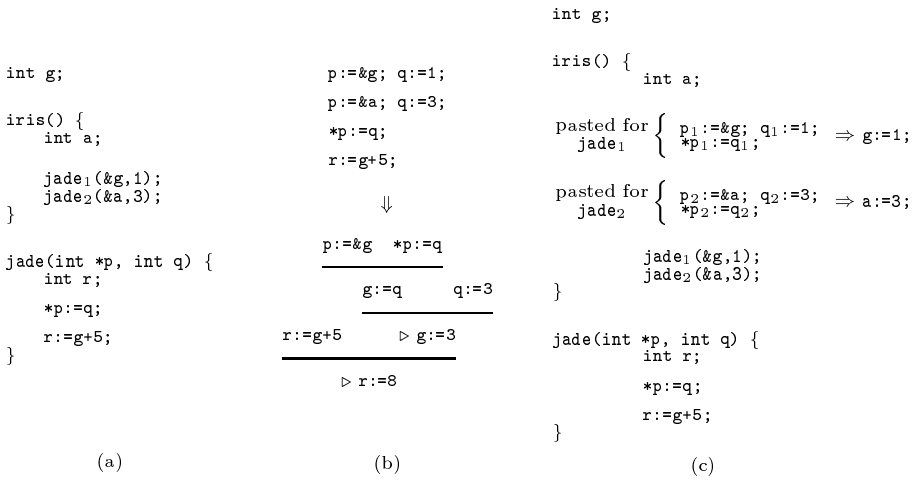

Fig. 1. Example 1: (a) code fragment, (b) flow- and context-insensitive equivalent followed by a possible derivation where the assignments marked with ▷ are spurious due to context loss, (c) the copying and pasting of `jade`'s summary `*p:=q` prevents the spurious derivation of `g:=3` and `a:=1`.

3. Procedural side effects are completely specialized. Thus, given an acyclic call graph, the implemented analysis is as accurate as one that transitively inlines all procedure calls.

The remainder of this paper is organized as follows. Section 2 demonstrates the limitations of previous approaches using a simple example. In Section 3, we give an overview how the proposed analysis overcomes those limitations followed by more detailed algorithmic descriptions in Section 4. Section 5 presents empirical results. In this paper, all the algorithms are presented as bottom-up logic programs [22]. In each inference rule, the information above the line compose the conditions in which the information below the line is derived. The actual algorithm only tracks pointer values, thus, ignores integer constants. However, in examples, we use integer constants as pointer surrogates for clarity.

2 Limitations with previous work

Through a simple example, this section illustrates the limitations of previous summary-based analyses. At the end of this section, the same example is used to demonstrate how the proposed analysis overcomes such limitations.

Example 1. The code fragment in Figure 1(a) shows typical pointer use for a C program. Within this example, the discussion focuses on the following data flow facts:

1. In `iris`, after the first call to `jade` (denoted $\texttt{jade}_1$), global variable `g` acquires `1`. After the second call to `jade` (denoted $\texttt{jade}_2$), variable `a` acquires `3`
2. In both calls to procedure `jade`, since global variable `g` always evaluates to `1`, local variable `r` always acquires `6`.

The top of Figure 1(b) shows a flow- and context-insensitive equivalent of Example 1, where the assignments from `iris` and `jade` are collected without procedural boundaries and ignoring the ordering among them. The following derivation in Figure 1(b) depicts an application of the inference rules to these assignments. When Example 1 is analyzed context-insensitively, the interaction of `p:=&g` from call site $\texttt{jade}_1$ and `q:=3` from call site $\texttt{jade}_2$ produces two spurious results `g:=3` and `r:=8`. Therefore, to capture the data flow facts in Example 1 accurately, context sensitivity is critical.

2.1 Bottom-up phase

In a typical summary-based analysis, the goal of the bottom-up phase is to construct a concise *flow function* for each procedure. Given a calling context as an input, this function should return the final pointer information of the procedure, incorporating all the effects of its callees.

All the flow functions are computed in a single bottom-up sweep of the call graph, propagating *procedure summaries* from callees to callers. The procedure summary is similar to the flow function except that, instead of the final pointer information for the whole procedure, it returns only the change in pointer information visible to the callers, called *procedural side effects*.

In Example 1, the only assignment in `jade` that causes side effects is `*p:=q`. Thus, the summary of `jade` consists of the single assignment `*p:=q`. In the bottom-up phase, as shown in Figure 1(c), a specialized version of `*p:=q` is pasted into each call site of `jade` (for instance, $\texttt{*p}_1\texttt{:=q}_1$ for $\texttt{jade}_1$) along with assignments that mimic parameter passing.

2.2 Top-down phase

The goal of the top-down phase is to compute the actual pointer information. Since the flow functions of each procedure are available, it can be computed in a single top-down sweep of the call graph. The top-down phase begins by analyzing procedure `iris`. As shown in 1(c), the inclusion of `jade`'s summary makes the effects of $\texttt{jade}_1$ and $\texttt{jade}_2$ exist locally. At `iris`, only the intended results `g:=1` and `a:=3` are derived leaving the pointer information for `iris` accurate and complete.

To analyze the next procedure, `jade`, the calling contexts must be formed then propagated (copied) from `iris` to `jade`. The calling contexts for the two calls to `jade` are:

$\texttt{jade}_1 \Rightarrow$ `g:=1; a:=3; p:=&g; q:=1;` $\quad\quad$ $\texttt{jade}_2 \Rightarrow$ `g:=1; a:=3; p:=&a; q:=3;`

To analyze procedure `jade` only once during the top-down phase, these calling contexts must be merged before the application of `jade`'s flow function, as shown in Figure 2(a). While bottom-up context sensitivity allowed the accurate derivation of `g`'s and `a`'s values in `iris`, the lack of top-down context sensitivity leads, again, to the spurious derivations `g:=3`, `a:=1`, and `r:=8`. Without intervention,

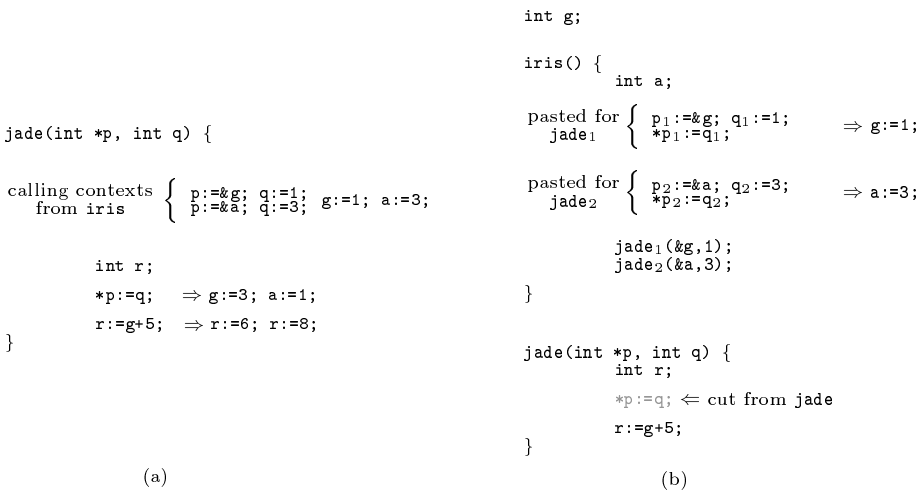

Fig. 2. Example 1 continued: (a) in the top-down phase for `jade` the spurious results `g:=3` and `r:=8` are still derived, (b) after cutting out the unnecessary assignment `*p:=q` from `jade`, the top-down phase no longer derives spurious results.

at the completion of `jade` all benefits of the context sensitive bottom-up phase have been lost.

In practice, despite of its negative impact on scalability, explicit copying of calling contexts during the propagation of pointer information from callers to callees contains some of the contamination. When the pointer information of `a` is needed, pointer analysis clients will look up the output of `iris`' flow function, which does not contain the spurious result `a:=1`. However, without true top-down context sensitivity, pointer analysis clients must querying `jade` for `r` and will obtain the spurious result `r:=8`.

2.3 Observation

The derivation of `r:=8` in Figure 2(a) can be avoided by removing `*p:=q` from `jade` following the construction of `jade`'s summary. After the bottom-up phase, the original assignment `*p:=q` in `jade` is irrelevant from the perspective of `iris`, since the pasting of `jade`'s summary provides local, specialized copies. Since `g`'s result is naturally available in `jade` through the calling context, the original assignment `*p:=q` in `jade` is unnecessary, and, more importantly, problematic.

This observation is reflected in Figure 2(b), where `*p:=q` has been removed from `jade`. After its removal, only the intended results are derived for both `iris` and `jade`. Moreover, since `iris` is truly unaffected by `jade` after this change, `iris`'s pointer information does not need to be set aside through explicit copying. The rest of this paper presents a generalization of the process of converting a program with procedural side effects to an equivalent one without any.

3 Overview

In C programs, indirect calls can be made using function pointers. In the presence of indirect calls, a cyclic dependency exists between call graph construction and pointer analysis. Without having a complete call graph, pointer information may be incomplete. On the other hand, without complete pointer information, the call graph may be incomplete.

The proposed analysis breaks this cyclic dependency using an iterative approach as in [3, 16]. It begins with an incomplete call graph consisting of only direct calls. Based on this incomplete call graph, pointer information is constructed. Then, using this as feedback, the call graph is updated. This process continues until there are no more changes to the call graph. Since the bottom-up phase does not terminate in the presence of recursive procedures, they are merged into a single procedure, while the recursive calls between them are converted into a set of parameter-passing assignments. Effectively, recursion is handled context-insensitively.

3.1 Impact of side effects

The rationale behind the proposed pointer analysis is best demonstrated by examining the impact of *procedural side effects* on the accuracy of context-insensitive analysis. Let us reconsider the example in Figure 1. By analyzing this example context-insensitively, we can derive following results.

$$\dfrac{\dfrac{[\texttt{p:=\&g}]_1 \quad \texttt{*p:=q}}{\texttt{g:=q} \qquad [\texttt{g:=1}]_1}}{\texttt{g:=1}} \qquad \dfrac{\dfrac{[\texttt{p:=\&a}]_2 \quad \texttt{*p:=q}}{\texttt{a:=q} \qquad [\texttt{q:=3}]_2}}{\texttt{a:=3}} \qquad \dfrac{\dfrac{[\texttt{p:=\&g}]_1 \quad \texttt{*p:=q}}{\texttt{g:=q} \qquad [\texttt{q:=3}]_2}}{\triangleright\ \texttt{g:=3}}$$

The first two derivations are from call sites 1 and 2 respectively and are both realizable in some dynamic execution of the code fragment. However, intermixing the parameter assignments from these two call sites results in the third, spurious, derivation reproduced from Figure 1(b). The key problem in this derivation stems from the fact that `g:=q` is valid only within the call site $\texttt{jade}_1$ while `q:=3` only within $\texttt{jade}_2$. Since they are from two different call sites, the two should not be allowed to interact.

As proved in the technical report [18], no such spurious interaction can occur if a program completely lacks procedural side effects. Even though a procedure may have many input calling contexts, without procedural side effects there is no way in which the contexts can interact. For side-effect free programs, full context sensitivity is inherent even when using a context-insensitive analysis.

3.2 Specialization of side effects

In reality, most programs have procedural side effects. Therefore, to exploit this finding, the proposed analysis hoists *specialized* copies of the procedural side

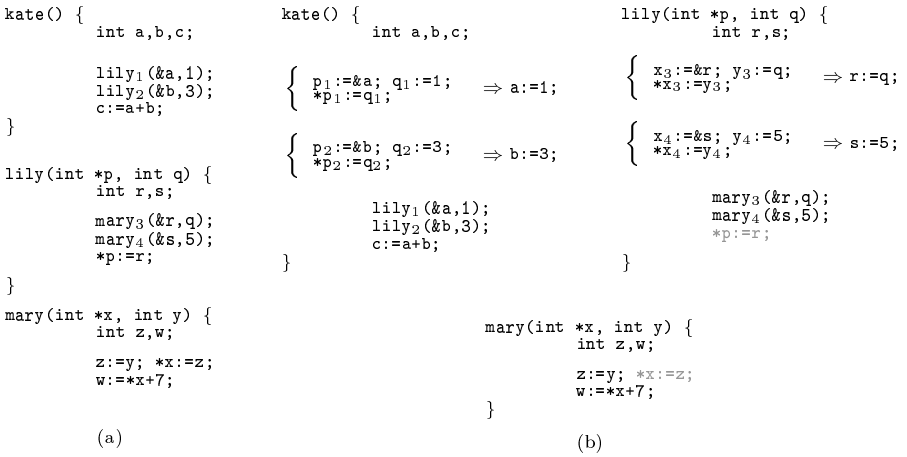

Fig. 3. Example 2: (a) code fragment, (b) demonstration of the specialization of procedural side effects.

effects from callees into callers and, then, cuts the side effects from the callees. The specialization continues until the program is free of procedural side effects thereby allowing the use of a context-insensitive style analysis in the following phase while avoiding all spurious interprocedural interaction.

Example 2. The code fragment in Figure 3(a) will be used to illustrate the key aspects of the specialization process in the proposed analysis.

1. In procedure `mary`, the assignment `*x:=z` is the direct cause of side effects. By specializing `*x:=z` into `mary`'s call sites (pasting its copies into call sites while cutting the original one), `mary` is left free of side effects, as shown in Figure 3(b).
2. Procedure `lily` also has side effects (due to `*p:=r`). Note that, when procedure `lily` is visited, the copies of `mary`'s summary are already present in `lily`. For this example, *compaction* (or simplification) is important to reduce the size of `lily`'s summary.

The bottom-up phase of our analysis traverses the acyclic-rendered call graph in a reverse-topological order, specializing side effects along the way. First, all the potential side-effect derivations are identified based on the concept of *criticality*. Informally speaking, we say that an assignment is *critical* if it has the potential to cause a side effect. In Example 2, the assignments `*x:=z` in `mary` and `*p:=r` in `lily` fall into this category.

The role of a procedure *summary* is to promote all the necessary side effects to the caller's level where the contexts will no longer intermix. Critical assignments are used as a seed in the creation of the summary. The pasting of summaries permits *cutting* out critical assignments from procedures, leaving them free of side effects while maintaining the effect of the original assignments.

Summary size is the single most important factor governing the scalability of this pointer analysis algorithm. In an extreme case, an entire callee can be used as a summary. However, considering their transitive impact on scalability,

it is necessary for summary sizes to be kept as small as possible. For instance, in `mary`, two assignments `*x:=z` and `z:=y` can be *compacted* into the single assignment `*x:=y` reducing the size of the summary. In addition, since the assignment `w:=*x+7` causes only local effects, by ignoring it, one can further reduce the summary size. *Back-tracing* is used to construct compacted assignments to be added into the summary. The following describes the actual back-tracing process in `mary` and the key intermediate decisions made during the process.

1. Back-tracing is initiated from the critical assignment `*x:=z`. However, since its right-hand side `z` is not a parameter, the addition of `*x:=z` into the summary is deferred and each assignment that modifies `z` is examined.
2. The assignment `z:=y` is the only assignment modifying variable `z`. Thus, by back-substituting `z` in `*x:=z` with `y`, the data flow is compacted into `*x:=y`. Since both `x` and `y` are parameters, no further compaction is necessary. Therefore, `*x:=y` is added to `mary`'s summary.

After the summary `*x:=y` is formed, it is pasted into each call site in `lily` while the critical assignment `*x:=z` is cut from `mary`.

3.3 Coping with aliases

For correctness, back-tracing must consider *all* data flow making it necessary to determine every location in which a variable may be modified. In reality, aliasing complicates this determination because complete data flow information is no longer explicit within the program text. The final decision on the form of the summary must be differed to higher level and will be formalized in Section 4. Consider procedure `lily` following the processing of `mary`.

1. Since the assignment `*p:=r` has the potential to modify an external variable, to construct `lily`'s summary, back-tracing is initiated from the assignment.
2. Note that no explicit data flow exists into `r`. However, since `r`'s address was taken at x_3`:=&r`, there may be implicit data flow that modifies `r`. In this case, all the data flow of `r` can be determined through a local examination on `lily`, resulting in the derivation of the implicit data flow `r:=q` from the specialized copy of `mary`'s summary for the call site $\texttt{mary}_3$.
3. By continuing back-tracing, `*p:=r` and the implicit data flow `r:=q` are compacted into `*p:=q`. Since there is no other modification of `r`, the assignment `*p:=q` suffices as `lily`'s summary.

After `lily` is processed, all side effects in Example 2 can be removed while leaving the overall pointer behavior unchanged. Then, the second phase applies a single run of the context-insensitive analysis and computes all pointer information completely and accurately. Note that, during this step, all the effects of the original critical assignments, namely, `*x:=z` in `mary` and `*p:=r` in `lily`, exist at the caller's level and are propagated back into the callees via regular parameter-passing mechanisms. After specialization, `mary`'s parameter `x` still points to `r` and `s`. Thus, `*x` correctly evaluates to `1`, `3`, and `5` in the assignment `w:=*x+7`.

An important aspect in `lily` that simplifies summary generation is that the aliases of the address-taken variables `r` and `s` could be completely and accurately

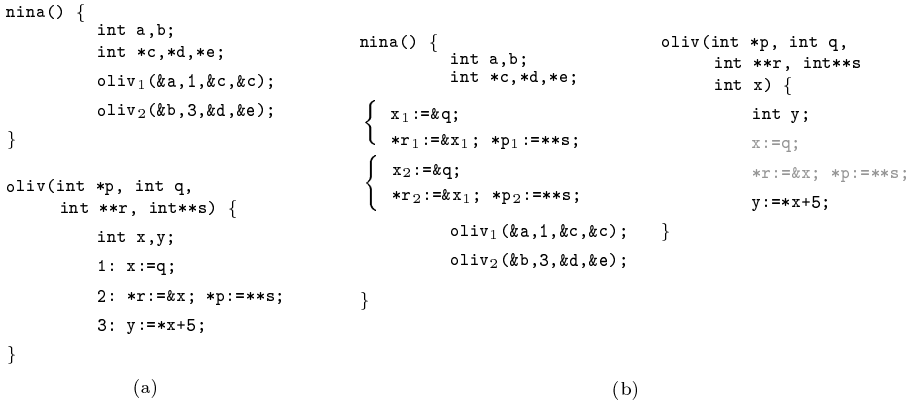

Fig. 4. Example 3: (a) code fragment before and (b) after side effects are specialized.

determined without knowledge of `lily`'s calling context. Unfortunately, this is not always possible.

Example 3. The code fragment in Figure 4(a) depicts a situation where accurate alias relations may not be determined without the knowledge of the actual calling contexts. The following are the key aspects of the example that complicate the specialization process.

1. In `oliv`, by the assignment `*r:=&x`, the address of local variable `x` is assigned to variables nonlocal to `oliv`. Note that this is a safe use of `x` since it is used only when `oliv` is active.
2. In the first call site $\texttt{oliv}_1$, since `r` and `s` point to the same variable `c`, `x` is copied into `*p` at Line 2. Therefore, there is a flow from `q` through `x` into `*p`, allowing `a:=1`.
3. In the second call site $\texttt{oliv}_2$, since `r` and `s` point to distinct variables, `x` is not copied into `*p` at Line 2. Instead, the contents of `*e` is read. Therefore, there is no flow from `q` to `*p`, thus `b` does not acquire `3`.

The last two aspects raise a dilemma. For the first call site $\texttt{oliv}_1$, the flow `*p:=q` must be reflected in the summary. However, for the second call site $\texttt{oliv}_2$, its inclusion would degrade the solution's accuracy. The problem is that the data flow in the summary depends on the calling contexts which are not available during the bottom-up phase.

There are multiple approaches to dealing with this dilemma. First, the assignment `*p:=q` can conservatively be included in the summary. Despite of accuracy degradation, this approach is sound. Second, two versions of summaries can be explicitly provided: one version for the case `*r` and `*s` are aliases and the other version for the case they are not. The summaries in [2] fall into this category.

Instead, in this paper, we take another approach. As shown in Figure 4(b), by introducing `x:=q`, `*r:=&x`, and `*p:=**s` into the summary, the alias of `x` can be resolved at the caller's level. Note that, to do so, `&x` and all potential writes to `x` have been included into the summary.

This decision has an impact on summary size. In an extreme case, where every variable behaves like `x`, the summary size will explode rapidly. However, in C programs, the lifetime of local variables is bounded by their declaring procedure and this constraint greatly influences typical programming practice. For this reason, the address of a local variable is rarely copied into a non-local memory space (which might be safe) or returned (which would be a bug).

The promotion of variable `x` from the callee `oliv` to the caller `nina` has another consequence. As shown in Figure 4(b), the assignment `y:=*x+5` accesses variable `x`, but only as a consumer. Thus, it causes only local effects and did not become a part of `oliv`'s summary. In order to keep the overall information unchanged, especially for variable `y`, we merge all specialized versions of variable `x` ($\mathtt{x}_1$ and $\mathtt{x}_2$) by treating them as extra arguments at the corresponding call sites as shown in Figure 4(b), Note that, after the merging, the following run of the context-insensitive analysis computes the pointer information completely and accurately.

4 Specialization algorithm

This section presents the specifics surrounding the identification of critical assignments and the assembly of concise summaries. The proposed analysis removes procedural side effects of a program by hoisting side effects upward in the call graph. Side effects are *specialized* until they can no longer interfere spuriously. When a procedure is about to be processed, all the side effects of its callees are already locally available in the procedure while the callees are left free of side effects. Therefore, in processing an individual procedure, we need to be concerned only with intraprocedural data flow.

4.1 Detecting criticality

As shown in the example in Figure 3, identifying critical assignments is the key to identifying procedural side effects. The difficulty in finding critical assignments lies in the fact that, when a procedure is being processed during the bottom-up traversal, no information about its calling contexts is available.

This difficulty is resolved by capturing the effects of calling contexts through two properties $\mathit{holding}, \mathit{opaque} : \mathbf{Var} \to \mathbf{Bool}$. Roughly speaking, a variable is said to be *holding* if it may point to an *external* variable. Provided that such information is available, critical assignments can be found by the rules in Figure 5.

Example 4. In the following code fragment, the parameter `p` is copied into the local variable `b` by the assignment `*a:=p`. Therefore, since `b` may point to an external variable, the assignment `*b:=r` is critical. Using the holding property described in Figure 5(a), the proposed analysis determines the criticality of `*b:=r` as shown in the following derivation.

$$\frac{u{:=}{*}v \quad holding(v)}{holding(u)}$$

$$\frac{u{:=}v \quad holding(v)}{holding(u)}$$

$$\frac{{*}u{:=}v \quad v{:=}\&w \quad holding(u)}{opaque(w)}$$

$$\frac{u{:=}\&v \quad opaque(u)}{opaque(v)}$$

$$\frac{param(u) \vee opaque(u)}{holding(u)}$$

$$\frac{u{:=}\&v \quad opaque(v)}{holding(u)}$$

(a)

$$\frac{{*}u := v \quad holding(u)}{critical({*}u := v)}$$

$$\frac{u := v \quad opaque(u)}{critical(u := v)}$$

$$\frac{u := {*}v \quad opaque(u)}{critical(u := {*}v)}$$

$$\frac{u := \&v \quad opaque(u)}{critical(u := \&v)}$$

(b)

Fig. 5. Symbolic derivation of side effects: (a) symbolic simulation of the effects of calling contexts, (b) determination of criticality.

```
pete(int *p, int q) {
    int *a,b;
    a:=&b; *a:=p; *b:=q;
}
```

$$\frac{\dfrac{\dfrac{\texttt{a:=\&b} \quad \texttt{*a:=p}}{\texttt{b:=p}} \quad \dfrac{param(p)}{holding(p)}}{\texttt{*b:=q} \qquad holding(\texttt{b})}}{critical(\texttt{*b:=q})}$$

In certain cases, the address of a local variable can be copied into an external variable. In the proposed analysis, this local variable is called *opaque* in the sense that its actual data flow becomes opaque due to the lack of calling contexts. This was seen in Example 4 where variable `x` in `oliv` is opaque. For a similar reason, a variable pointed to by an opaque variable must be regarded as being opaque as well. Note that, to be conservative, an opaque variable must be regarded as holding, too.

In back-tracing, an opaque variable is problematic because its alias information is unknown making it unclear where or how it is modified. As explained in Section 3.3, we handle this ambiguity by promoting the relevant assignments into the callers thereby hoisting the alias resolution to a point where more information about the calling contexts is known. This implies that *all* potential modifications of such variables must be reflected in the summary. From this perspective, as depicted by the rules in Figure 5(b), it is convenient to treat an opaque variable as external.

$$\frac{traced(*u := v) \quad u := w \quad holding(w)}{traced(*w := v)}$$

$$\frac{traced(*u := v) \quad u := *w \quad holding(w)}{add(*u := v) \wedge traced(u := *w)}$$

(a) stuck left-hand

$$\frac{traced(*u := v) \quad v := w \quad holding(w)}{traced(*u := w)}$$

$$\frac{traced(*u := v) \quad v := \&w}{add(*u := v) \wedge traced(v := \&w)}$$

$$\frac{traced(*u := v) \quad v := *w \quad holding(w)}{add(*u := v) \wedge traced(v := *w)}$$

(b) stuck right-hand

$$\frac{traced(*u := v) \quad input(u) \quad input(v)}{add(*u := v)}$$

(c) stuck both sides

$$\frac{traced(u := v) \quad v := \&w}{traced(u := \&w)}$$

$$\frac{traced(u := v) \quad v := *w \quad holding(w)}{traced(u := *w)}$$

$$\frac{traced(u := v) \quad v := w \quad holding(w)}{traced(u := w)}$$

$$\frac{traced(u := v) \quad input(v)}{add(u := v)}$$

$$\frac{traced(u := \&v)}{add(u := \&v)}$$

(d)

$$\frac{traced(u{:=}*v) \quad v{:=}w \quad holding(w)}{traced(u{:=}*w)}$$

$$\frac{traced(u := *v) \quad v := *w \quad holding(w)}{add(u := *v) \wedge traced(v := *w)}$$

$$\frac{traced(u := *v) \quad input(v)}{add(u := *v)}$$

(e)

Fig. 6. Back-tracing for (a)(b)(c) store, (d) plain and address, (e) and load assignments.

4.2 Back-tracing

Given critical assignments, including those due to opaque variables, summarization begins by initiating *back-tracing* as follows, where e and e' stand for arbitrary expressions:

$$\frac{critical(e{:=}e')}{traced(e{:=}e')}$$

Though the back-tracing rules used by our analysis (shown in Figure 6) appear complicated, the invariants behind them are straightforward: once an assignment is *traced*, all its effects must be reproduced by the summary. Actual insertion of a back-traced assignment into the summary is deferred until it becomes unavoidable and will occur due to one of the following conditions: First, back-tracing has reached an *input* to the procedure (parameter or opaque variable). Second, the net effect from two assignments cannot be represented with a single assignment in the model language (a single edge in the actual implementation). For instance, `u:=**v` is not allowed, thus, must be broken into `u:=*t` and `t:=*v`.

5 Empirical results

To demonstrate the usefulness of our approach, we evaluated the presented techniques on all the C benchmarks from the SPEC 92, 95, and 2000 suites except

Benchmark	LOC	CI	CS		R%
			BU	TD	
008.espresso	13505	1.00	0.30	0.03	89 %
023.eqntott	3393	0.02	0.02	0.01	- %
099.go	28547	0.50	0.15	0.02	- %
124.m88ksim	17251	0.70	0.40	0.04	4 %
129.compress	1426	0.01	0.01	0.01	- %
130.li	6930	8.00	2.00	0.03	- %
132.ijpeg	25897	3.50	35.00	1.00	8 %
134.perl	23969	2.50	30.00	1.50	10 %

Benchmark	LOC	CI	CS		R%
			BU	TD	
164.gzip	7759	0.02	0.01	0.01	2 %
175.vpr	16973	0.25	0.10	0.01	35 %
176.gcc	205747	45.00	190.00	2.50	2 %
181.mcf	1909	0.01	0.02	0.01	- %
186.crafty	18977	0.10	0.07	0.01	33 %
197.parser	10924	0.40	0.15	0.02	- %
253.perlbmk	57541	360.00	800.00	45.00	4 %
254.gap	59674	120.00	430.00	18.00	1 %
255.vortex	52634	6.00	10.00	1.00	- %
256.bzip2	4637	0.01	0.01	0.01	- %
300.twolf	19749	0.30	0.15	0.02	- %

Fig. 7. Analysis time in seconds for the context insensitive (CI) and context sensitive (CS) analyses. The CS time is divided into the bottom-up (BU) and top-down (TD) portions. It also shows the percentage reduction of total points-to set size in percentage (the column R%) when switching from CI to CS. Times were measured on a 2.4 GHz Pentium 4 computer and never consumed more than 300 MB of memory.

Benchmark	Proposed			Foster *et al.*		
	BU	TD	Ratio	BU	TD	Ratio
008.espresso	0.31	0.03	0.10	28.81	967.64	33.6
023.egntott	0.02	0.01	0.50	1.50	11.20	7.5
129.compress	0.01	0.01	1.00	0.41	1.42	3.5
130.li	1.96	0.03	0.02	189.49	9929.88	52.4

Fig. 8. Analysis time in seconds compared to Foster *et al.* [12]. Ratio is the ratio of TD time to BU time. There is a clear difference in ratios, with the TD phase for the proposed analysis only contributing to a small percentage of the total time.

redundant ones. This breadth of benchmarks shows a wide range of analysis characteristics. The main results are summarized in Table 7.

Since field information can greatly affect the analysis results, the analysis is implemented in a field-sensitive fashion. Our implementation tries to handle as many abuses of the C language as possible in a safe and accurate manner. For instance, our analysis is offset-based, as opposed to using field names as in [19]. Each variable is associated with *accessed offsets*, which are dynamically updated should any additional offsets be discovered during the analysis.

To reduce summary size, the bottom-up phase performs redundancy elimination [17] and cycle elimination [9]. In addition, data flow through global variables are treated context-insensitively, which allows their exclusion from procedural summaries while not impacting resultant accuracy.

Table 7 lists the percentage reduction in total points-to set size when comparing context-insensitive (CI) results to context-sensitive (CS) results. This is calculated by summing the points-to set size for every location (variable offset) in the benchmark. While this may not directly correlate to beneficial accuracy improvement, we feel it provides useful comparison. There is a substantial range in benefit with some benchmarks showing little or no benefit (256.bzip and 300.twolf) while others show a fairly substantial reduction (008.espresso at 89% and 175.vpr at 35%).

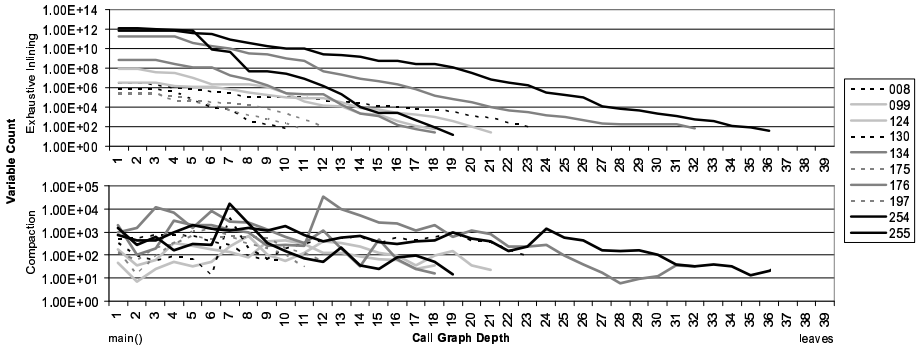

Fig. 9. Semi-log graph showing the number of variable locations versus call graph depth for both exhaustive inlining and our proposed summary-based inlining across SPEC.

Table 7 lists two sets of analysis times (in seconds) for CI and CS analysis runs. The CS run times are broken into bottom-up and top-down times. When comparing the CI and CS analysis times, there is a large span ranging from the cases where the CS times being faster (008.espresso, 130.li, 175.vpr, and 197.parser) to the other cases where the CS times being about 5-10x slower (132.ijpeg, 134.perl, and 254.gap). It is interesting to note that there is a strong correlation between the CI and CS accuracy and analysis time comparison. The more accuracy benefit the CS analysis provided, the faster the analysis performs with respect to the CI analysis.

It is also enlightening to compare the relative cost of the bottom-up and top-down phases. Aside from those benchmarks that require less than about 0.1 second, the top-down phase of the analysis contributes little to the overall analysis time. It is generally between 1% and 10% of the bottom-up time. This is expected, because, while the bottom-up phase spends time summarizing procedures and specializing procedural side effects, the top-down phase only needs to propagate pointer information in a context-insensitive fashion.

To provide a frame of reference concerning the efficiency of our top-down phase, Figure 8 presents top-down and bottom-up analysis times from the benchmarks in common between our work and Foster *et al.* [12]. The important comparison is the ratio of top-down to bottom-up time for each of two works. For our analysis, the top-down time is never more than the bottom-up time, even for a benchmark as small as 129.compress, and becomes increasingly insignificant as benchmark size increases (2% for 130.li). We suspect that the copying overheads are what cause the top-down phase in Foster *et al.* to generally surpass the bottom-up time, reaching a factor of 50 for 130.li.

One major concern in terms of scalability is the problem size growth caused by the inclusion of specialized summaries. Figure 9 demonstrates the effectiveness of our bottom-up phase by comparing the theoretical growth in exhaustive inlining (upper graph) against the empirical results using our summary-based analysis (lower graph). In the figure, procedure `main` is always at a depth of one while leaf procedures are at the far right of a particular line spanning depths from 11 to 39. The y-axis is a logarithmic scale of the number of variables at

that level. Thus, the values in the upper graph extend to 1 trillion while the lower graph extends only to 100K.

It is apparent that the growth in our bottom-up phase is not explosive. An interesting aspect of the lower graph is that the size of summaries does not grow monotonically as it approaches `main`. Instead, it varies almost independently of the depth, increasing and decreasing according to compaction opportunities. The only spikes in the lower graph are due to the existence of large recursive cycles (100s of procedures). Even for these, summarization quickly diminishes their impact.

6 Related work

Pointer analysis has been studied extensively in the literature. For a more complete list of previous pointer analyses, we refer readers to [15]. Context-sensitive pointer analyses have been approached from several directions. In [8, 16, 19, 25, 27], procedures are re-analyzed until a global fixed point is reached. Among these, [27] is unique in the sense that the overhead of re-analysis is reduced by memoization. On the other hand, the analyses in [2, 3, 12, 20, 21] are based on procedure summaries formulated as two-phase computation. Among those, Foster *et al.* [12] is the closest to the proposed pointer analysis. One of the analyses presented by Foster *et al.* is a context-sensitive extension of Andersen's analysis. The bottom-up phase of this algorithm is similar to ours in many aspects. The difference is that, in Foster *et al.*, the critical assignments in callees still exist after the bottom-up phase. Therefore, in addition to accuracy degradation, explicit copying seems unavoidable.

CFL reachability [24] provides an alternative approach to context sensitivity. One of its merits over other approaches is that recursion can be handled also context-sensitively. Fähndrich *et al.* [10] employed a variant of CFL reachability with unification-based modeling proposed by Steensgard [26]. Das *et al.* [7] took a similar approach using the data flow modeling proposed by Das [6] yet with one-level context sensitivity.

The philosophy behind our pointer analysis is similar to *counter-example directed refinement*, a popular scheme in model checking [5]. In this approach, refinement is driven by feedback from a less accurate abstraction. In our pointer analysis, instead, refinement is performed preemptively and exhaustively. Guyer and Lin [13] applied counter-example directed refinement to pointer analysis. When results computed by less accurate pointer analysis (both flow- and context-insensitive) turn out to be insufficient, the accuracy is refined by adding flow or context sensitivity. To add context sensitivity, a procedure call is specialized by cloning the callee's body. After being specialized, all the procedures are re-analyzed until a global fixed point is reached. From this perspective, the techniques proposed in this paper can also greatly aid the efficiency of their analysis.

Many algorithms, for instance, [11, 12], employ techniques similar to back-tracing in an effort to construct a *concise* yet *observably-equivalent* summary of the analysis information. Constraint simplification in general has been thoroughly studied. For a more complete list of references and in-depth discussion, we

refer readers to [23]. The concept of opacity is similar to the *escaping* of objects extensively discussed for object-oriented programs. For the list of references, we refer readers to [4].

7 Conclusion

We have proposed a fully context-sensitive yet very efficient summary-based pointer analysis. The key aspect is that the bottom-up phase transforms a program into one lacking procedural side effects by cutting side-effect causing assignments from callees after they have been summarized. This prevents all spurious interprocedural data flow in the following top-down phase while preserving the pointer behavior. Empirical results support the effectiveness of the pointer analysis both in accuracy and efficiency. We thank the IMPACT research group for their help and DARPA/MARCO-GSRC which supported this work.

References

1. L. O. Andersen. *Program analysis and specialization for the C programming language.* Ph.D thesis, DIKU, Unversity of Copenhagen, 1994.
2. R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *POPL*, 1999.
3. B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, 2000.
4. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *TOPLAS*, 2003.
5. E. Clarke, O. Grunmberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 1997.
6. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, 2000.
7. M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, 2001.
8. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, 1994.
9. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, 1998.
10. M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.
11. C. Flanagan and M. Felleisen. Componential set-based analysis. In *PLDI*, 1997.
12. J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS*, 2000.
13. S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *SAS*, 2003.
14. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *PLDI*, 2001.
15. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.
16. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *TOPLAS*, 21(4):848–894, 1999.

17. J. E. Hopcroft. An $n\log n$ algorithm for minimizing the states of a finite automaton. *The Theory of Machines and Computations*, pages 189–196, 1971.
18. H.-S. Kim, E. M. Nystrom, and W. W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. Technical report, IMPACT research group, University of Illinois. 2004.
19. W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *PLDI*, 1992.
20. C. Lattner and V. Adve. Data structure analysis: a fast and scalable context-sensitive heap analysis. Technical report, CS Dept., University of Illinois. 2003.
21. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *FSE*, 1999.
22. D. A. McAllester. On the complexity analysis of static analysis. In *SAS*, 1999.
23. J. Rehof. Minimal typings in atomic subtyping. In *POPL*, 1997.
24. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
25. E. Ruf. Context-insensitive alias analysis reconsidered. In *PLDI*, 1995.
26. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
27. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.