SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2214

Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation

Yulei Sui[1], Sen Ye[1], Jingling Xue[1,*,†] and Jie Zhang[2]

[1]*School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia*
[2]*College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, China*

SUMMARY

Because of its high precision as a flow-insensitive pointer analysis, Andersen's analysis has been deployed in some modern optimising compilers. To obtain improved precision, we describe how to add context sensitivity on top of Andersen's analysis. The resulting analysis, called ICON, is efficient to analyse large programs while being sufficiently precise to drive compiler optimisations. Its novelty lies in summarising the side effects of a procedure by using one transfer function on virtual variables that represent fully parameterised locations accessed via its formal parameters. As a result, a good balance between efficiency and precision is made, resulting in ICON that is more powerful than a 1-callsite-sensitive analysis and less so than a call-path-sensitive analysis (when the recursion cycles in a program are collapsed in all cases). We have compared ICON with FULCRA, a state of the art Andersen's analysis that is context sensitive by acyclic call paths, in Open64 (with recursion cycles collapsed in both cases) using the 16 C/C++ benchmarks in SPEC2000 (totalling 600 KLOC) and 5 C applications (totalling 2.1 MLOC). Our results demonstrate scalability of ICON and lack of scalability of FULCRA. FULCRA spends over 2 h in analysing SPEC2000 and fails to run to completion within 5 h for two of the five applications tested. In contrast, ICON spends just under 7 min on the 16 benchmarks in SPEC2000 and just under 26 min on the same two applications. For the 19 benchmarks analysable by FULCRA, ICON is nearly as accurate as FULCRA in terms of the quality of the built Static Single Assignment (SSA) form and the precision of the discovered alias information. Copyright © 2013 John Wiley & Sons, Ltd.

Received 14 October 2012; Revised 21 June 2013; Accepted 27 June 2013

KEY WORDS: pointer analysis; inclusion-based analysis; context-sensitive analysis

1. INTRODUCTION

Pointer analysis is critical to enable advanced and aggressive compiler optimisations. Andersen's inclusion-based analysis [1] is a highly precise pointer analysis, which is flow insensitive (by ignoring control flow) and context insensitive (by ignoring calling contexts). With its recent advances, Andersen's analysis, which is more precise than Steensgaard's unification-based analysis [2], is now scalable for large programs [3]. In the latest release of the Open64 compiler, its pointer analysis is no longer unification based but rather inclusion based, performed with offset-based field sensitivity and 1-callsite-sensitive heap cloning (with malloc wrappers being recognised as heap allocation sites). Just like GNU GCC, the overall pointer analysis framework in Open64 remains context insensitive. However, many compiler optimisations benefit, in both precision and effectiveness, from more precise points-to information if context sensitivity is also considered. Unfortunately, existing context-sensitive versions of Andersen's analysis are not scalable to millions of lines of code. To the best of our knowledge, there is presently no suitable context-sensitive Andersen's analysis that can be deployed in modern compilers such as Open64 and GCC.

*Correspondence to: Jingling Xue, Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia.

†E-mail: jingling@cse.unsw.edu.au

In this paper, we introduce a whole-program context-sensitive Andersen's analysis for C/C++ programs, called ICON, that scales to millions of lines of code. ICON is significantly faster than the state of the art while achieving nearly the same precision. While implemented fully in Open64, our analysis applies to any inclusion-based framework.

The development of ICON has been guided by three design principles:

Precision For an optimising compiler, its pointer analysis must soundly estimate the points-to information in a program. As far as performance improvements are concerned, it will be unnecessarily costly to obtain context sensitivity if the context of a call is identified by its *full* call path. Instead, we prefer to find a faster solution that may not be theoretically powerful but is practically precise enough in driving compiler optimisations. Our measurement of precision is the quality of the built SSA form (in terms of χ and μ operations introduced [4]) and the percentage of aliases disambiguated. We believe that these two metrics are critical in determining the effectiveness of compiler optimisations, such as register allocation [5], instruction scheduling [6], redundancy elimination [7–9], scratchpad management [10–13] and speculative parallelisation [14–17].

Efficiency Context sensitivity should be achieved on top of Andersen's analysis with as little overhead as possible for large programs.

Simplicity The solution should be simple conceptually and implementation-wise. To this end, some recent advances in inclusion-based analysis should be leveraged so that the existing code base is maximally reused. Specifically, we prefer to achieve context sensitivity by staying in the same inclusion-based analysis framework.

1.1. The state of the art

A context-insensitive pointer analysis does not distinguish between different invocations of a procedure. When analysing a program, passing parameters and return values between procedures is modelled as assignments without distinguishing their calling contexts. Some precision loss is illustrated in Figure 1. In Figure 1(a), the information from one caller is allowed to flow into another. So both `x` and `y` may point to `a` and `b`. In Figure 1(b), the information from the two callsites is merged at the entry of `foo` so that `*p` and `*q` are considered to alias in both calling contexts. As a result, `r` is regarded as pointing to `g` always even though this is possible only when `foo` is called inside `goo2`.

A context-sensitive pointer analysis improves precision by representing calling contexts of a procedure more accurately. An analysis is *k-callsite context sensitive* if different invocations of a

```
1  void goo() {
2     int a, b;
3     int *x, *y;
4     x = foo(&a);
5     y = foo(&b);
6  }
7
8  int* foo(int *p) {
9     return p;
10 }
```

(a)

```
1  int g;
2  void goo1() {
3     int *a, *b;
4     foo(&a, &b);
5  }
6  void goo2() {
7     int *c;
8     foo(&c, &c);
9  }
10
11 void foo(int **p, int **q) {
12    int *r;
13    *p = &g;
14    r = *q;
15 }
```

(b)

Figure 1. Imprecision in context-insensitive analysis.

DOI: 10.1002/spe

procedure are distinguished by k enclosing callsites in its calling contexts. In practice, the contexts of a method call are often represented by their acyclic call paths, with recursion cycles collapsed or unrolled k times. If k represents the maximal length of acyclic call paths, then the context sensitivity is achieved with full call paths. In Figure 1, precise points-to sets can be obtained with $k = 1$.

For Java (which relies on heap-only object allocation), much progress has been made in context-sensitive points-to analysis [18–31]. However, we have not seen a corresponding success for C and C++ due to their support for pointer operations such as address-of operator & for both heap and stack objects and pointer arithmetic. Many flow-sensitive and/or context-sensitive analyses [32–40] have been proposed. According to [41], however, current industrial-strength flow-sensitive and context-sensitive versions of Andersen's analysis are scalable only for small C/C++ programs. They are not deployable yet in an optimising compiler. Even if flow sensitivity is ignored, how to achieve context sensitivity efficiently and precisely on top of Andersen's analysis for whole C/C++ programs remains to be an open problem. Among some earlier attempts at making Andersen's (flow insensitive) analysis context sensitive [42–45], FULCRA [43] represents a state-of-the art solution. By cloning (conceptually) the statements in a procedure that may have interprocedural points-to side effects and then inlining them directly in its callers, FULCRA obtains the most precise points-to information as an inclusion-based analysis by being context sensitive with acyclic call paths (i.e. with recursion cycles collapsed). In obtaining such cloning-based precision, however, FULCRA does not scale to some large programs [43].

1.2. Our insights

Our key observation is that real-world C/C++ programs are likely to be dominated by procedures with a small number of pointer formal parameters, which are each dereferenced with a few levels of indirection. Figure 2 plots some pointer-related information among the formal parameters of the procedures in the 21 programs used in our experiments, including the 15 C and one C++ benchmarks from SPEC2000 (totalling 600 KLOC) and five applications (totalling 2.1 MLOC). Most procedures (92.5% on average) have fewer than four PFPs. Among the procedures with two (three) PFPs, most of their PFPs, with an average of 89.8% (91.9%), have fewer than four levels of indirection. This suggests that in the code written by programmers, the formal parameters of a procedure at its entry tend to have few and simple aliasing relations.

This observation has led to the design of our ICON analysis. Its novelty lies in exploiting parameterised pointer information to achieve context sensitivity on top of Andersen's analysis. For a procedure being analysed, ICON represents the abstract locations passed from its callers and accessed by its dereferenced formal parameters using virtual variables and keeps track of their aliasing relations during the analysis. This enables the interprocedural points-to side effects of a procedure to be summarised with a transfer function that maps each virtual variable to its points-to set. Each points-to relation is guarded by an aliasing condition on virtual variables so that context sensitivity can be achieved when it is 'transferred' to its callsites. By using virtual variables rather than individual locations, the propagation of points-to information is significantly accelerated.

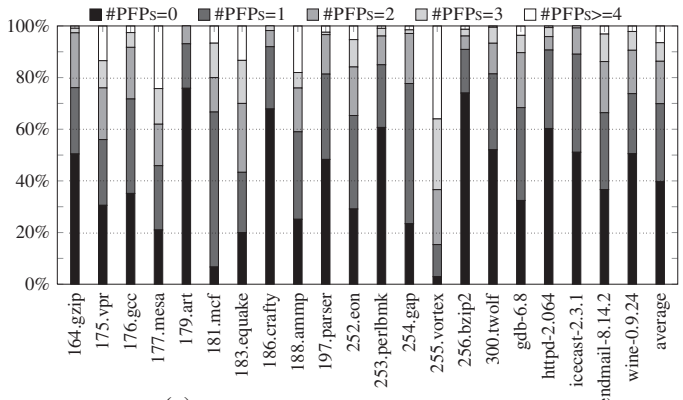
In theory, ICON is more powerful than a 1-callsite-sensitive analysis but less so than a cloning-based context-sensitive analysis such as FULCRA (when the recursion cycles in a program are collapsed in all cases). In practice, ICON is significantly faster than FULCRA while achieving nearly the same precision. In addition, ICON is simple as it can be implemented easily on top of Andersen's analysis, which is widely used with industrial-strength implementations available.

1.3. Contributions

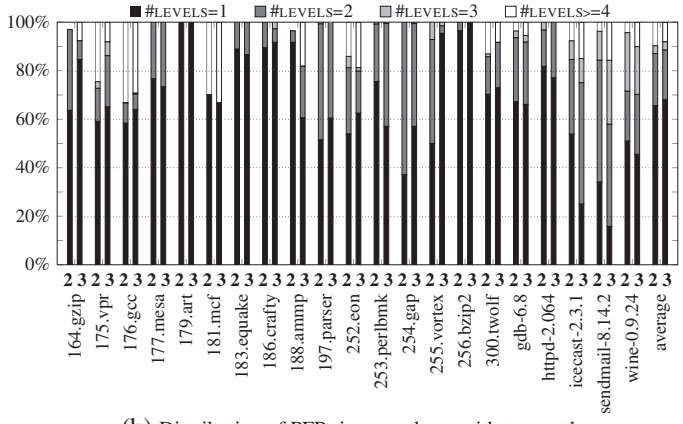
While symbolic names [35,39,40,46,47] and transfer functions [39,40] were previously used, ICON exploits both in a novel way to obtain a scalable context-sensitive Andersen's analysis.

- We introduce a context-sensitive Andersen's inclusion-based pointer analysis, ICON, that can be directly and easily deployed in modern optimising compilers. While existing solutions are not scalable, ICON, which is fully implemented in Open64, allows large programs (with millions of lines of code) to be analysed in minutes.

DOI: 10.1002/spe



(a) Distribution of procedures according to the number of **pointer** formal parameters (PFPs)



(b) Distribution of PFPs in procedures with two or three PFPs according to the number of levels of indirection

Figure 2. Pointer-related information for formal parameters using Andersen's analysis.

- ICON is the first to achieve context sensitivity on top of Andersen's analysis by computing the transfer function for a procedure based on parameterised pointer information in terms of virtual variables, motivated by the pointer-related information at the procedure entries in real-world C/C++ programs (Figure 2). In addition, we also propose to accelerate ICON by pre-analysis (to discover non-aliased parameters) and by propagating points-to information first top-down and then bottom-up on the call graph of a program (to discover aliased parameters eagerly).
- We have evaluated ICON by comparing with FULCRA, a state-of-the-art Andersen's analysis that is context sensitive by acyclic call paths [43], in Open64 (with recursion cycles collapsed in both cases) using 21 C/C++ programs, including 15 C and 1 C++ benchmarks in SPEC2000 (600 KLOC) and 5 C applications (2.1 MLOC). Our results demonstrate scalability of ICON and lack of scalability of FULCRA for some large programs. FULCRA spends over 2 h in analysing SPEC2000 and fails to run to completion within 5 h for two of the five applications tested, `wine` and `gdb`. In contrast, ICON spends only just under 7 min on SPEC2000 and just under 26 min on both `wine` and `gdb`. For the 19 benchmarks analysable by FULCRA, ICON is nearly as accurate as FULCRA in terms of the quality of the built SSA form and the precision of the discovered alias information.

The rest of this paper is organised as follows. Section 2 provides some more background information. Section 3 motivates ICON with an example used throughout the paper. Section 4 introduces

DOI: 10.1002/spe

the ICON analysis. Section 5 evaluates ICON and compares it with the state of the art. Section 6 discusses related work, and Section 7 concludes the paper.

2. BACKGROUND

We first describe the canonical representation used for a program and then introduce Andersen's analysis by constraint resolution.

2.1. Program representation

Four types of statements are considered: $x = \&y$ (address), $x = *y$ (load), $*x = y$ (store) and $x = y$ (copy). Note that $x = ** y$ can be transformed into $x = *t$ and $t = *y$ by introducing a new temporary t. Different fields of a struct are distinguished, but arrays are considered monolithic.

Every call contained in a procedure *goo* has the form $r = foo(a_1, \ldots, a_n)$, where $r, a_1, \ldots, a_n$ are local variables in *goo* and *foo* is a callee (resolved on the fly during pointer analysis). As ICON is context sensitive, we must keep track of the (modification) side effects made by *foo* on the variables accessed in *goo*. For efficiency considerations, the modification side effects on the global variables made in all procedures are tracked globally in the standard manner as in [43, 47]. In contrast, the non-global variables accessed in *goo* may be modified in *foo* in two ways: (1) via the formal parameters of *foo* and (2) by passing a return value to *goo*. Such modification side effects are referred to as the side effects of *foo* in this paper and tracked by using a transfer function for *foo*. In order to deal with these two types of side effects on non-globals uniformly, we perform a standard transformation as shown in Figure 3 so that (2) can be dealt with equivalently as (1).

2.2. Constraint-based Andersen's analysis

As illustrated in Figure 4, Andersen's analysis discovers points-to information by treating assignments as subset constraints using a single constraint graph for the entire program until a fixed point is reached. When context sensitivity is not considered, passing parameters and return values between procedures is simply modelled as copy statements.

For the code in Figure 4(a), Andersen's analysis starts with the constraint graph given in Figure 4(b). For the address statements, `y=&x` and `n=&g`, the points-to information is directly recorded for their left-hand side variables. For each of the other three types of statements, a constraint of an appropriate type is introduced. Then, the analysis resolves loads and stores by adding new copy statements discovered. As `y` points to `x`, the two new copy statements related to x are added as shown in Figure 4(c). The new points-to information discovered is propagated along the edges in the graph until a fixed point is found. Finally, `t` is found to point to `g`.

```
void goo(···) {
    int *a1, ..., *an, *r;
    ...
    r = foo(a1, ..., an);
    ...
}

int* foo(int *f1, ..., int *fn) {
    int *s;
    ...
    return s;
}
```

(a)

```
void goo(···) {
    int *a1, ..., *an, *r;
    ...
    foo(&r, a1, ..., an);
    ...
}

void foo(int **r', int *f1, ..., int *fn) {
    int *s;
    ...
    *r' = s;
}
```

(b)

Figure 3. Passing return values modelled as passing parameters.

DOI: 10.1002/spe

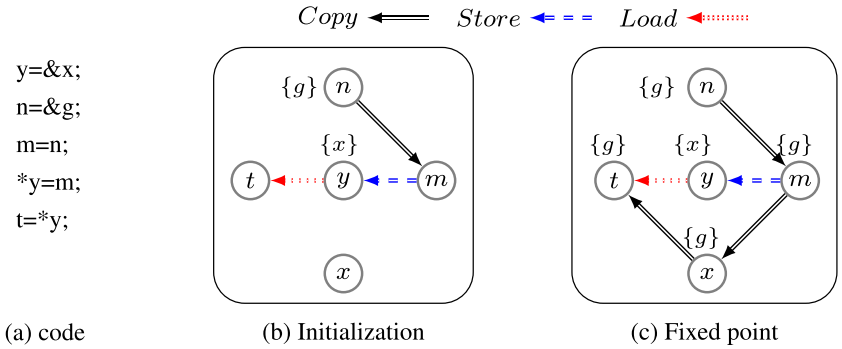


Figure 4. Pointer resolution in a constraint graph.

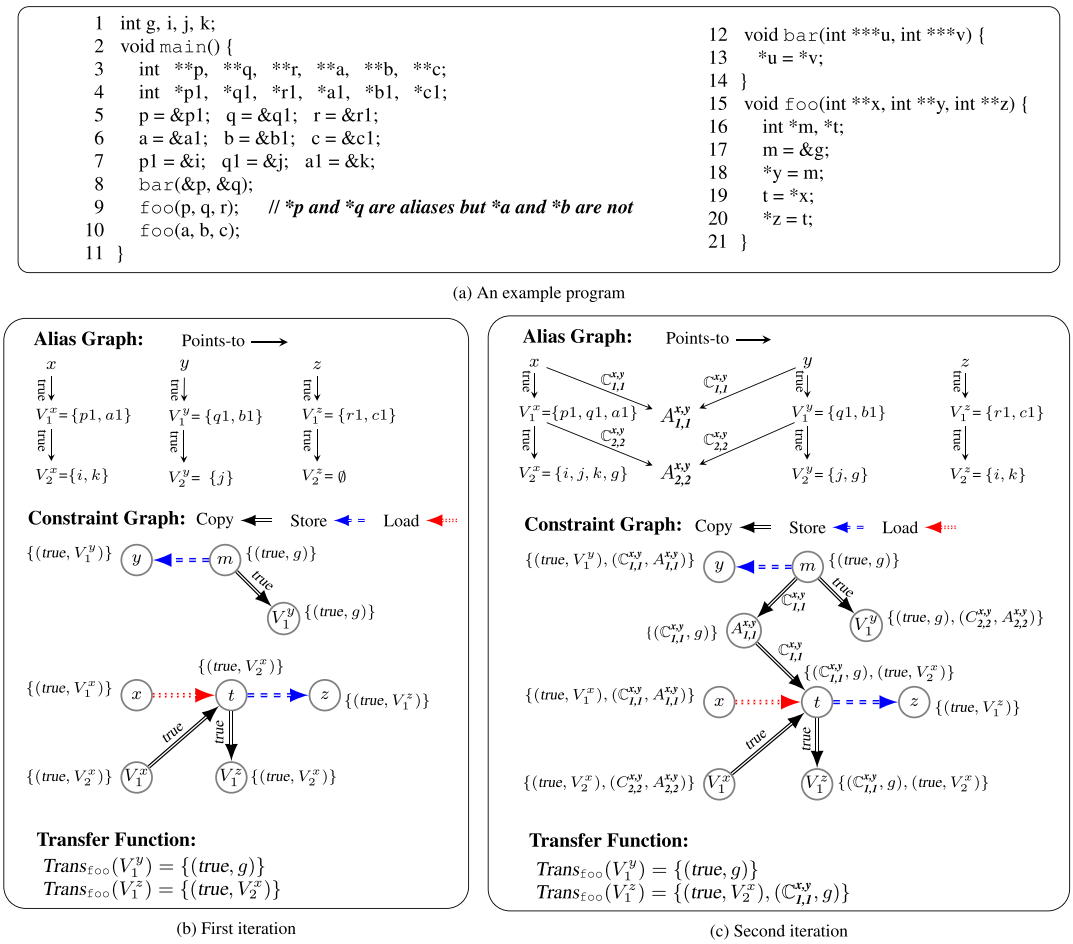


Figure 5. Parameterised summarisation for `foo` with virtual variables. The points-to set for a node is not completely shown, but it can be read off as the set of all objects reaching the node by copy edges.

3. A MOTIVATING EXAMPLE

We use an example as shown in Figure 5 to illustrate how we achieve context sensitivity on top of Andersen's analysis. The key novelty is to summarise the side effects of a procedure in terms of virtual variables that represent fully parameterised pointer information at its entry. Guided by the

DOI: 10.1002/spe

three design principles discussed earlier in Section 1, ICON is developed to discover precise points-to information efficiently in a simple way by leveraging the same constraint resolution engine used by Andersen's analysis.

In the program given in Figure 5(a), `*p` and `*q` are made aliases but `*a` and `*b` are not aliased after the call to `bar`. Thus, after the two calls to `foo`, `g` will be pointed to by `r1` but not by `c1`. We examine how this fact is discovered by ICON. In a context-insensitive analysis, however, `g` will be conservatively estimated as being pointed to by both `r1` and `c1`.

ICON analyses a program by traversing its call graph (with recursion cycles collapsed) iteratively, first top-down and then bottom-up. In a top-down phase, the points-to information is propagated downwards from a caller to its callees, with the side effects of all callsites being ignored. In a bottom-up phase, the propagation of the points-to information is reversed from a callee to its callers, with the points-to side effects of the callee being summarised and transferred to its callsites. In both phases, new points-to information is discovered in the same constraint resolution framework.

We focus on how `foo` is summarised and how its summarised side effects are transferred to its two callsites. The call graph comprises a root node, `main`, and its two child nodes, `bar` and `foo`.

3.1. First iteration

Top-down The analysis starts with `main` and then moves to `bar` and `foo`. The points-to relations in lines 5–7 in `main` are discovered trivially and propagated downwards into `bar` and `foo`. In our analysis, the points-to relations from different callsites in a procedure are merged and represented using an alias graph at its entry but handled (at least 1-callsite) context sensitively. At this stage, the one for `foo` is given in Figure 5(b). V_1^f and V_2^f, where $f \in \{\texttt{x}, \texttt{y}, \texttt{z}\}$, stand for $*f$ and $**f$, respectively. These are *virtual variables*, each of which represents the set of non-local locations passed from the two callsites of `foo` and accessed by the dereferenced parameters in `foo`.

Each procedure has its own constraint graph except that copy edges are guarded, stating the conditions under which the corresponding relations hold. In the special case when a copy edge is guarded by *true*, the corresponding relation always holds. Andersen's analysis is applied to the constraint graphs of all procedures combined, except that (1) the side effects of all callsites are ignored and (2) virtual variables are used to parameterise pointer information at procedure entries. In the case of `foo`, the initial constraint graph (not shown) comprises (1) the constraints corresponding to the statements in lines 17–20 and (2) the points-to relations in its alias graph. After the fixed-point is reached, we obtain the constraint graph given in Figure 5(b).

Bottom-up Andersen's analysis is applied *separately* to the constraint graphs of different procedures. The interprocedural points-to side effects of a procedure are summarised and transferred to its callsites. In the case of `foo`, there is no need to re-run Andersen's analysis as no new points-to information is discovered. The transfer function of `foo` that maps $V_1^{\texttt{y}}$ and $V_1^{\texttt{z}}$ to their points-to sets as shown is obtained. Note that `x` is not modified inside. Similarly, the transfer function of `bar` (not shown) is computed: $Trans_{\text{bar}}(V_1^{\texttt{u}}) = \{(true, V_2^{\texttt{v}})\}$. When `main` is analysed, the side effects of `foo` on $V_1^{\texttt{y}}$ and $V_1^{\texttt{z}}$ are transferred to its two callsites. For the first callsite, $V_1^{\texttt{y}}$ and $V_1^{\texttt{z}}$ stand for `q1` and `r1`, respectively. So `q1` has a new target `g` and `r1` a new target `i`. For the second callsite, $V_1^{\texttt{y}}$ and $V_1^{\texttt{z}}$ stand for `b1` and `c1`, respectively. So `b1` has a new target `g` and `c1` a new target `k`. Similarly, applying `bar`'s transfer function to its callsite, we find that `p` has a new target `q1`.

3.2. Second iteration

Top-down Some new points-to relations that have been discovered in the first iteration are propagated downwards. At this stage, `foo`'s alias graph is the same as that in Figure 5(b)

DOI: 10.1002/spe

except that the contents of its virtual variables have been updated with the new points-to information, as shown in Figure 5(c). In addition, the constraint graph for `foo` remains the same as the one in Figure 5(b).

Bottom-up When `foo` is analysed, its alias graph is inspected. At this stage, the alias graph is the same as the one shown in Figure 5(c) except that the four points-to relations, $\mathtt{x} \xrightarrow{\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}} A^{\mathtt{x},\mathtt{y}}_{1,1}$, $\mathtt{y} \xrightarrow{\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}} A^{\mathtt{x},\mathtt{y}}_{1,1}$, $V^{\mathtt{x}}_1 \xrightarrow{\mathbb{C}^{\mathtt{x},\mathtt{y}}_{2,2}} A^{\mathtt{x},\mathtt{y}}_{2,2}$ and $V^{\mathtt{y}}_1 \xrightarrow{\mathbb{C}^{\mathtt{x},\mathtt{y}}_{2,2}} A^{\mathtt{x},\mathtt{y}}_{2,2}$, do not exist yet. Once some new points-to information is available in an alias graph, ICON will proceed to identify and establish all the new aliasing relations between virtual variables and update the alias graph accordingly. In the case of `foo`, $V^{\mathtt{x}}_1$ and $V^{\mathtt{y}}_1$ are found to alias. In addition, $V^{\mathtt{x}}_2$ and $V^{\mathtt{y}}_2$ are also found to alias. To represent these new aliasing relations, the four aforementioned points-to relations are introduced. Here, $\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}$ is an aliasing condition that encodes $V^{\mathtt{x}}_1 \cap V^{\mathtt{y}}_1 \neq \emptyset$. Similarly, $\mathbb{C}^{\mathtt{x},\mathtt{y}}_{2,2}$ encodes $V^{\mathtt{x}}_2 \cap V^{\mathtt{y}}_2 \neq \emptyset$. $A^{\mathtt{x},\mathtt{y}}_{1,1}$ ($A^{\mathtt{x},\mathtt{y}}_{2,2}$) symbolises the set of aliased locations between $V^{\mathtt{x}}_1$ and $V^{\mathtt{y}}_1$ ($V^{\mathtt{x}}_2$ and $V^{\mathtt{y}}_2$). Such sets are placeholders as their contents are not directly used during pointer resolution. Hence, the contents of $A^{\mathtt{x},\mathtt{y}}_{1,1}$ and $A^{\mathtt{x},\mathtt{y}}_{2,2}$ are not shown in Figure 5(c). It is the presence of aliasing relations such as $\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}$ and $\mathbb{C}^{\mathtt{x},\mathtt{y}}_{2,2}$ that serves to enable points-to information to be propagated conditionally across the aliased locations.

Re-propagating the new points-to information just introduced across the constraint graph for `foo` in Figure 5(b) yields the fixed point given in Figure 5(c). As a result, the transfer function for `foo` is updated from the one given in Figure 5(b) to the one given in Figure 5(c). During this second round of guarded constraint resolution, the two new copy edges added for $A^{\mathtt{x},\mathtt{y}}_{1,1}$ are guarded by $\mathbb{C}^{\mathtt{xy}}_{1,1}$. This indicates that $V^{\mathtt{z}}_1$ points to `g` only when $\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}$ holds.

By applying `foo`'s transfer function to its first callsite at line 9, we find that $\mathcal{M}^{\mathtt{foo}}_9(\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}) = (\mathcal{M}^{\mathtt{foo}}_9(V^{\mathtt{x}}_1) \cap \mathcal{M}^{\mathtt{foo}}_9(V^{\mathtt{y}}_1) \neq \emptyset) = (\{\mathtt{q1}\} \neq \emptyset) = \mathit{true}$, where $\mathcal{M}^{\mathtt{foo}}_9(V^{\mathtt{x}}_1) = \{\mathtt{p1}, \mathtt{q1}\}$ and $\mathcal{M}^{\mathtt{foo}}_9(V^{\mathtt{y}}_1) = \{\mathtt{q1}\}$ include only the locations in $V^{\mathtt{x}}_1$ and $V^{\mathtt{y}}_1$ propagated from the first callsite. Thus, `j` and `g` are new targets of `r1`. For the second callsite at line 10, $\mathcal{M}^{\mathtt{foo}}_{10}(\mathbb{C}^{\mathtt{x},\mathtt{y}}_{1,1}) = (\mathcal{M}^{\mathtt{foo}}_{10}(V^{\mathtt{x}}_1) \cap \mathcal{M}^{\mathtt{foo}}_{10}(V^{\mathtt{y}}_1) \neq \emptyset) = (\emptyset \neq \emptyset) = \mathit{false}$, where $\mathcal{M}^{\mathtt{foo}}_{10}(V^{\mathtt{x}}_1) = \{\mathtt{a1}\}$ and $\mathcal{M}^{\mathtt{foo}}_{10}(V^{\mathtt{y}}_1) = \{\mathtt{b1}\}$ contain only the locations propagated from the second callsite. Therefore, no new points-to relations are found at the second callsite, implying that `g` cannot be pointed to by `c1`. This mapping process is explained in more detail in Example 5.

4. THE ICON ALGORITHM

As motivated by our example, the earlier ICON discovers the aliasing information at procedure entries, the earlier it can find the points-to sets for more pointers, and consequently, the faster the analysis converges. Therefore, its three components are structured as shown in Algorithm 1.

ALGORITHM 1: The ICON algorithm

```
1 Pre-Analysis
2 repeat
3 |   Top-Down Analysis
4 |   Bottom-Up Analysis
5 until a fixed point is reached;
```

Pre-analysis discovers non-aliased formal parameters and initialises the alias graphs for all procedures. During a top-down phase, the points-to information in a program is propagated top-down across its acyclic call graph, with all recursion cycles collapsed on the fly. During a bottom-up phase, the direction of points-to information propagation is reversed. During each iteration, the top-down

DOI: 10.1002/spe

phase precedes the bottom-up phase so that procedure pointers can be resolved as soon as possible as suggested in [43]. In ICON, doing so has an additional benefit: the aliasing relations among the formal parameters of a procedure can be discovered as soon as possible.

Each procedure has its own constraint graph. A top-down phase processes the constraint graphs of all procedures together while a bottom-up phase deals with each individually. In both phases, guarded constraint propagation is used. ICON achieves context sensitivity by maintaining alias graphs during the analysis, causing the side effects of a procedure to be summarised iteratively.

Section 4.1 discusses the guarded constraint resolution used in both top-down and bottom-up phases. Section 4.2 introduces pre-analysis. Section 4.3 focuses on top-down analysis and Section 4.4 on bottom-up analysis. Section 4.5 discusses some salient properties of ICON.

4.1. Guarded constraint propagation

Much progress has been made on efficiently solving the inclusion-based constraints for pointer analysis [3, 45, 48–50]. We extend a recent pointer resolution algorithm, which is adapted from wave propagation [50] and implemented by the Open64 team in the Open64 compiler, to perform the guarded constraint resolution in ICON (in the presence of virtual variables).

The rules used by SOLVECONSTRAINTS are given in Table I. The notation $\mathsf{ptr}(p)$ stands for the points-to set of a variable p. The notation $\mathsf{ptr}(a) \supseteq_c \mathsf{ptr}(b)$ means that for each $(c', o) \in \mathsf{ptr}(b)$, its propagation into a is conditional so that $(c \wedge c', o) \in \mathsf{ptr}(a)$. In ICON, each pointed-to object is guarded. When resolving a load or a store, all copy edges derived are guarded accordingly.

Example 1
When moving from Figure 5(b) to (c), two copy edges are added. When resolving `*y = m` in line 18, `y` points to $(\mathbb{C}^{x,y}_{1,1}, A^{x,y}_{1,1})$. So $A^{x,y}_{1,1} \overset{\mathbb{C}^{x,y}_{1,1}}{\Longleftarrow} \mathtt{m}$ is added. When resolving `t = *x` in line 19, `x` points to $(\mathbb{C}^{x,y}_{1,1}, A^{x,y}_{1,1})$. As a result, $t \overset{\mathbb{C}^{x,y}_{1,1}}{\Longleftarrow} A^{x,y}_{1,1}$ is added.

4.2. Pre-analysis

Steensgaard's unification-based analysis [2] is used to bootstrap ICON. By interpreting assignments as equality rather than subset constraints, Steensgaard's analysis is less precise but significantly faster than Andersen's analysis. Pre-analysis serves two purposes. First, many formal parameters that do not alias, as observed in Figure 2, are discovered, avoiding unnecessary aliasing tests later. Second, an alias graph for each procedure that is expressed in terms of virtual variables, as illustrated in our example, is initialised. The domain of a virtual variable, which denotes the set of locations that it represents in the subsequent analysis, is also determined.

Let us describe how to initialise an *alias graph*, $\mathsf{AG}_P = (V_P, E_P)$, for a procedure P. We focus on an arbitrary parameter f of P because all parameters are handled identically and independently. After Steensgaard's analysis, the *points-to graph* G_f for f, which comprises the points-to relations originating from f, is always a 'linked list' with at most one cycle at its end. As Steensgaard's analysis is used, all locations pointed by a variable are unified into the same equivalence class. For example, if the points-to relations originating from f are $f \to a$, $a \to b$ and $a \to c$, then G_f is $f \to \{a\} \to \{b, c\}$ with two equivalence classes. If the points-to relations are $f \to a$, $a \to b$, $b \to c$, $c \to b$ and $c \to d$ instead, then G_f becomes $f \to \{a\} \to \{b, c, d\}$↺ with a cycle at its end.

Table I. Rules used by guarded constraint resolution.

Statement	Constraint	Resolution
Address	$x = \&y$	$\{(true, y)\} \in \mathsf{ptr}(x)$
Copy	$x = y$	$\mathsf{ptr}(x) \supseteq_{true} \mathsf{ptr}(y)$
Load	$x = *y$	$\forall (c', y') \in \mathsf{ptr}(y) : x \supseteq_{c'} y'$
Store	$*x = y$	$\forall (c', x') \in \mathsf{ptr}(x) : x' \supseteq_{c'} y$

DOI: 10.1002/spe

Let $\mathsf{depth}(f)$ be the number of equivalence classes in the points-to graph G_f. $\mathsf{AG}_P = (V_P, E_P)$ is initialised as follows. For every parameter f of P, we add the $\mathsf{depth}(f) + 1$ nodes that represent f (which is V_0^f), $V_1^f, \ldots, V_{\mathsf{depth}(f)}^f$ and the $\mathsf{depth}(f)$ edges $f \xrightarrow{true} V_1^f$, $V_1^f \xrightarrow{true} V_2^f, \ldots, V_{\mathsf{depth}(f)-1}^f \xrightarrow{true} V_{\mathsf{depth}(f)}^f$. If G_f has a cycle (at its end), we also add $V_{\mathsf{depth}(f)}^f \xrightarrow{true} V_{\mathsf{depth}(f)}^f$.

$V_1^f, \ldots, V_{\mathsf{depth}(f)}^f$ are called *virtual variables*, each of which is used to represent the set of non-local locations passed from P's callers and accessed by P's dereferenced formal parameters during the analysis. Intuitively, V_i^f stands for $\overbrace{*\cdots*}^{} f$ with exactly i $*$'s if $i < \mathsf{depth}(f)$ and all $\overbrace{*\cdots*}^{} f$'s with $\mathsf{depth}(f)$ or more $*$'s if $i = \mathsf{depth}(f)$. Thus, each guarded edge thus introduced represents a points-to relation that always holds because the guard is true.

When G_f has a cycle, our parameterisation-based approach looks seemingly conservative because $V_{\mathsf{depth}(f)}^f$ represents $\overbrace{*\cdots*}^{} f$'s with $\mathsf{depth}(f)$ or more $*$'s. However, as suggested by the statistics given in Figure 2 and evaluated further in our experiments in Section 4.5, little precision is lost for real code.

Each virtual variable V_i^f is created with empty points-to information in pre-analysis but will be iteratively updated (or filled up) during the subsequent pointer analysis. The domain of V_i^f, denoted $\mathsf{dom}(V_i^f)$, is simply the i-th equivalence class in the points-to graph G_f, with i starting from 1.

Example 2
$\mathsf{AG}_{\texttt{foo}}$ is initialised as shown in Figure 5(b), except that all the virtual variables are empty initially. After Steensgaard's analysis, the points-to graphs for $G_{\texttt{x}}$, $G_{\texttt{y}}$ and $G_{\texttt{z}}$ are $\texttt{x} \rightarrow e_1 \rightarrow e_2$, $\texttt{y} \rightarrow e_1 \rightarrow e_2$ and $\texttt{z} \rightarrow e_3 \rightarrow e_2$, where $e_1 = \{\texttt{p1,q1,a1,b1}\}$, $e_2 = \{\texttt{i,j,k,g}\}$ and $e_3 = \{\texttt{r1,c1}\}$. Thus, each formal parameter is associated with two virtual variables. In addition, $\mathsf{dom}(V_1^{\texttt{x}}) = \mathsf{dom}(V_1^{\texttt{y}}) = e_1$, $\mathsf{dom}(V_2^{\texttt{x}}) = \mathsf{dom}(V_2^{\texttt{y}}) = e_2$, $\mathsf{dom}(V_1^{\texttt{z}}) = e_3$ and $\mathsf{dom}(V_2^{\texttt{z}}) = e_2$.

Lemma 1
Given a formal parameter f, V_i^f and V_j^f never alias with each other if i and j are different.

Proof
As each virtual variable represents an equivalence class created by Steensgaard's analysis, therefore, $\mathsf{dom}(V_i^f) \cap \mathsf{dom}(V_j^f) = \emptyset$. Hence, V_i^f and V_j^f do not alias. □

4.3. Top-down analysis

During a top-down phase, ICON resolves all the constraints in a program except that the side effects of all callsites are ignored. As a result, the points-to information propagated interprocedurally only flows from a caller to its callees. By summarising the side effects of a procedure using a transfer function in terms of virtual variables, this phase is surprisingly simple and efficient. Andersen's analysis is simply performed on the constraint graphs of all procedures in a program simultaneously except the guarded constraint resolution described in Section 4.1 is used. To disregard the side effects of a callee invoked at a callsite, we simply do not apply its transfer function at the callsite.

The new points-to information is propagated downwards from a caller into its callees. Let P be a procedure invoked at a callsite. Let f be a formal parameter of P and a the corresponding actual parameter at the callsite. Whenever some new points-to information for a is discovered, it is propagated into the virtual variables $V_1^f, \ldots, V_{\mathsf{depth}(f)}^f$ of f. For each target x pointed by a directly or indirectly such that $x \in \mathsf{dom}(V_i^f)$, two cases are distinguished. If x is a virtual variable, then x is *flattened* so that all the actual points-to targets represented by x are inserted into V_i^f. Otherwise, x itself is inserted into V_i^f.

DOI: 10.1002/spe

Example 3
Consider the top-down phase performed during the first iteration in Figure 5(b). The constraint graph for `foo` initially consists of the four constraints in lines 17–20 and those in its alias graph. In `main`, there are two callsites to `foo`. The points-to relations for their actual parameters are discovered in lines 5–7 in `main`. There are six virtual variables for `foo` with their domains given in Example 2. By propagating the new points-to relations discovered in `main` into these virtual variables, which are empty initially, we obtain `foo`'s alias graph in Figure 5(b). Note that `p` $\rightarrow$ `p1` $\rightarrow$ `i` and `a` $\rightarrow$ `a1` $\rightarrow$ `k` are propagated into $V_1^{\mathtt{x}}$ and $V_2^{\mathtt{x}}$, `q` $\rightarrow$ `q1` $\rightarrow$ `j` and `b` $\rightarrow$ `b1` into $V_1^{\mathtt{y}}$ and $V_2^{\mathtt{y}}$, and `r` $\rightarrow$ `r1` and `c` $\rightarrow$ `c1` into $V_1^{\mathtt{z}}$ and $V_2^{\mathtt{z}}$. Once Andersen's analysis (using guarded constraint resolution) is completed, the fixed-point found for `foo` is given in Figure 5(b).

Suppose we add a call `goo(x)` inside `foo`, where `goo(int **s) { ... }` is unspecified here. As `x` $\rightarrow$ $V_1^{\mathtt{x}}$, where $V_1^{\mathtt{x}} = \{\mathtt{p1}, \mathtt{a1}\}$, then `p1` and `a1` will be propagated into the virtual variable $V_1^{\mathtt{s}}$ of `goo` after the points-to target $V_1^{\mathtt{x}}$ is flattened because $V_1^{\mathtt{x}}$ is virtual.

During top-down analysis, the call graph of a program is updated as follows. First, the call graph is expanded on the fly whenever a new callee pointed by a procedure pointer is detected. Second, each recursion cycle, that is, strongly connected component (SCC), detected is collapsed so that all procedures contained inside are analysed context insensitively.

4.4. Bottom-up analysis

During a bottom-up phase, ICON resolves all the constraints in a program by accounting for the side effects of all callsites. As a result, the points-to information propagated interprocedurally flows upwards from a callee into its callers. Each SCC has its own constraint graph. Andersen's analysis is applied to these constraint graphs separately using our guarded constraint resolution.

We traverse the SCCs in a program's call graph in their reverse topological order. On visiting an SCC, we discover its new points-to relations parametrically in terms of the virtual variables at its entry. At the same time, its interprocedural points-to side effects on virtual variables are obtained iteratively. The entry of an SCC is formed by combining the entries of all procedures contained in the SCC admitting calls from outside the SCC. Thus, their alias graphs are naturally merged. For this reason, we shall speak of SCCs and procedures interchangeably below.

ALGORITHM 2: Bottom-up analysis

```
1 for each SCC P in the program's call graph in reverse topological order do
2 |   Stage 1: UpdateAliasGraph(P)
3 |   Stage 2: ApplyTransFun(P)
4 |   Stage 3: SolveConstraints(P)
  end
```

As shown in Algorithm 2, an SCC P is analysed in three stages. First, P's alias graph is updated to establish any new aliasing relations for its virtual variables (Section 4.4.1). Second, the interprocedural points-to side effects at P's callsites are accounted for (Section 4.4.2). Finally, a new round of guarded constraint propagation is started to discover more points-to relations for P, if needed (Section 4.4.3). Because of a cyclic dependence between stages 2 and 3, Section 4.4.3 can be read before Section 4.4.2 to ease understanding.

4.4.1. Stage 1: Updating alias graphs. After the preceding top-down phase in the current iteration is over, some virtual variables of a procedure P may contain new points-to targets. As a result, there may be new aliases formed among the virtual variables of P. Note that the virtual variables of the same parameter do not alias with each other by Lemma 1.

UpdateAliasGraph given in Algorithm 3 is simple. We examine every pair of parameters f and g to look for new aliasing relations (line 1). If f and g are discovered not to alias in pre-analysis, we are finished (lines 3 and 4). Otherwise, we proceed to discover new aliasing relations between

DOI: 10.1002/spe

ALGORITHM 3: UPDATEALIASGRAPH(P) //$\mathsf{AG}_P = (V_P, E_P)$

1 **repeat**
2 **for** *every pair of formal parameters* f *and* g *of* P **do**
3 **if** *NON-ALIAS*(f, g) **then**
4 **continue**
  end
5 **for** *every pair of virtual variables* V_i^f *and* V_j^g *in* V_P *(for some* i *and* j*) such that* $V_i^f \cap V_j^g \neq \emptyset$ **do**
6 **if** *there exists* $C(V_i^f, V_j^g) = V_i^f \cap V_j^g \neq \emptyset$ *encoded earlier in line 8* **then**
7 **continue**
  end
8 Encode $C(V_i^f, V_j^g)$ as $V_i^f \cap V_j^g \neq \emptyset$
9 Add a new *aliasing variable* $A_{i,j}^{f,g}$, which is a placeholder standing for $V_i^f \cap V_j^g$
10 Add $V_{i-1}^f \xrightarrow{C(V_i^f, V_j^g)} A_{i,j}^{f,g}$ and $V_{j-1}^g \xrightarrow{C(V_i^f, V_j^g)} A_{i,j}^{f,g}$ where V_0^f denotes f and V_0^g denotes g by the construction of an alias graph (when $i = 1$)
  end
 end
11 **until** *no new aliases are found*;

V_i^f, a virtual variable of f, and V_j^g, a virtual variable of g (line 5). We introduce a new aliasing relation between V_i^f and V_j^g in P's alias graph (lines 8–10) if it is not encompassed by an existing one (lines 6–7). In line 9, $A_{i,j}^{f,g}$ symbolises $V_i^f \cap V_j^g$ but is not actually computed. Such *aliasing variables* serve as a conduit to allow points-to information to be propagated along aliased locations.

Example 4
Let us apply `UpdateAliasGraph` to `foo` during the second iteration illustrated in Figure 5(c). Initially, `foo`'s alias graph is the same as that in Figure 5(b) except that the contents of all virtual variables are shown as in Figure 5(c). Because $V_1^{\mathtt{x}} \cap V_1^{\mathtt{y}} = \{\mathtt{q1}\}$, $\mathbb{C}_{1,1}^{\mathtt{x},\mathtt{y}}$ is introduced to encode $V_1^{\mathtt{x}} \cap V_1^{\mathtt{y}} \neq \emptyset$. By using $A_{1,1}^{\mathtt{x},\mathtt{y}}$ to symbolise $V_1^{\mathtt{x}} \cap V_1^{\mathtt{y}}$, two new points-to relations, $V_0^{\mathtt{x}} \xrightarrow{C(V_1^{\mathtt{x}}, V_1^{\mathtt{y}})} A_{1,1}^{\mathtt{x},\mathtt{y}}$ and $V_0^{\mathtt{y}} \xrightarrow{C(V_1^{\mathtt{x}}, V_1^{\mathtt{y}})} A_{1,1}^{\mathtt{x},\mathtt{y}}$, are introduced, where $V_0^{\mathtt{x}}$ and $V_0^{\mathtt{y}}$ stand for `x` and `y`, respectively. Similarly, because $V_2^{\mathtt{x}} \cap V_2^{\mathtt{y}} = \{\mathtt{j}, \mathtt{g}\}$, $\mathbb{C}_{2,2}^{\mathtt{x},\mathtt{y}}$ is introduced to encode $V_2^{\mathtt{x}} \cap V_2^{\mathtt{y}} \neq \emptyset$. By using $A_{2,2}^{\mathtt{x},\mathtt{y}}$ to symbolise $V_2^{\mathtt{x}} \cap V_2^{\mathtt{y}}$, two new points-to relations, $V_1^{\mathtt{x}} \xrightarrow{C(V_2^{\mathtt{x}}, V_2^{\mathtt{y}})} A_{2,2}^{\mathtt{x},\mathtt{y}}$ and $V_1^{\mathtt{y}} \xrightarrow{C(V_2^{\mathtt{x}}, V_2^{\mathtt{y}})} A_{2,2}^{\mathtt{x},\mathtt{y}}$, are introduced. As a result, `foo`'s alias graph has been updated from Figure 5(b) to (c).

It is easy to see that UPDATEALIASGRAPH terminates as the number of aliasing relations is finite. The following lemma states that all aliasing relations at the entry of a procedure are captured.

Lemma 2 (Aliasing)
Let f and g be two formal parameters of a procedure P. If V_i^f and V_j^g alias for some i and j, then there must exist an aliasing relation $\mathcal{C}$ and an aliasing variable $\mathcal{A}$ such that $V_{i-1}^f \xrightarrow{\mathcal{C}} \mathcal{A}$ and $V_{j-1}^g \xrightarrow{\mathcal{C}} \mathcal{A}$ in P's alias graph, where $\mathcal{C} = C(V_i^f, V_j^g)$ and $\mathcal{A} = A_{i,j}^{f,g}$.

Proof
Following from the construction of an alias graph given in Algorithm 3. □

DOI: 10.1002/spe

4.4.2. Stage 2: Applying transfer functions. Let us describe how to transfer context sensitively the side effects of every callee Q invoked at each callsite I that is contained in a procedure P. For a virtual variable v of Q, we write $\mathcal{M}_I^Q(v)$ to represent the subset of v that is propagated only from the callsite I. For an aliasing condition $c = v_i \cap v_j \neq \emptyset$ that appears in Q's alias graph, we overload $\mathcal{M}_I^Q$ by writing $\mathcal{M}_I^Q(c)$ to mean $\mathcal{M}_I^Q(v_i) \cap \mathcal{M}_I^Q(v_j) \neq \emptyset$, which represents the aliasing condition created at the callsite I alone (as if the other callsites were non-existent).

ALGORITHM 4: APPLYTRANSFUN(P)

```
1  for each callsite I in P do
2      for each callee Q invoked at callsite I do
3          for each virtual variable V_i^f of Q do
4              S ⟵ M_I^Q(V_i^f)
5              for each (c, v) ∈ Trans_Q(V_i^f) do
6                  if M_I^Q(c) holds then
7                      if v is a virtual variable V_j^g of Q then
8                          T ⟵ M_I^Q(V_j^g)
                       else
9                          T ⟵ {v}
                       end
10                     for (s, t) ∈ S × T do
11                         Insert (true, t) into s's points-to set in P's constraint graph
                       end
                   end
               end
           end
       end
   end
```

APPLYTRANSFUN given in Algorithm 4 is straightforward. The transfer function $Trans_Q$ referred to in line 5 is defined precisely in Section 4.4.3. It is worth emphasising that a guarded points-to relation (c, v) created in a callee Q is ignored at a callsite I when its guard $M_I^Q(c)$ evaluates to *false* at the callsite (line 6). As a result, the imprecision illustrated in Figure 1(b) is avoided.

Example 5

Consider Figure 5(c) again. Let us illustrate Algorithm 4 for `main` by applying $Trans_{\texttt{foo}}(V_1^{\texttt{z}}) = \{(true, V_2^{\texttt{x}}), (\mathbb{C}_{1,1}^{\texttt{x,y}}, \texttt{g})\}$ to its two callsites at lines 9–10 where `foo` is invoked. We have $P = \texttt{main}$, $Q = \texttt{foo}$ and $I \in \{9, 10\}$. In the first callsite at line $I = 9$, we know from Figure 5(c) that $\mathcal{M}_9^{\texttt{foo}}(V_1^{\texttt{z}}) = \{\texttt{r1}\}$ and $\mathcal{M}_9^{\texttt{foo}}(V_2^{\texttt{x}}) = \{\texttt{i}, \texttt{j}\}$. From the target $(true, V_2^{\texttt{x}})$, the points-to relations $\texttt{r1} \xrightarrow{true} \texttt{i}$ and $\texttt{r1} \xrightarrow{true} \texttt{j}$ are established at this callsite. For the other target $(\mathbb{C}_{1,1}^{\texttt{x,y}}, \texttt{g})$, we know that $\mathcal{M}_9^{\texttt{foo}}(\mathbb{C}_{1,1}^{\texttt{x,y}}) = (\mathcal{M}_9^{\texttt{foo}}(V_1^{\texttt{x}}) \cap \mathcal{M}_9^{\texttt{foo}}(V_1^{y}) \neq \emptyset) = (\{\texttt{p1}, \texttt{q1}\} \cap \{\texttt{q1}\} \neq \emptyset) = (\{\texttt{q1}\} \neq \emptyset) = true$. So the points-to relation $\texttt{r1} \xrightarrow{true} \texttt{g}$ is also found. For the second callsite at line $I = 10$, $\mathcal{M}_{10}^{\texttt{foo}}(V_1^{\texttt{z}}) = \{\texttt{c1}\}$ and $\mathcal{M}_{10}^{\texttt{foo}}(V_2^{\texttt{x}}) = \{\texttt{k}\}$. As $\mathcal{M}_{10}^{\texttt{foo}}(\mathbb{C}_{1,1}^{\texttt{x,y}}) = (\mathcal{M}_{10}^{\texttt{foo}}(V_1^{\texttt{x}}) \cap \mathcal{M}_{10}^{\texttt{foo}}(V_1^{\texttt{y}}) \neq \emptyset) = (\{\texttt{a1}\} \cap \{\texttt{b1}\} \neq \emptyset) = (\emptyset \neq \emptyset) = false$, only $\texttt{c1} \xrightarrow{true} \texttt{k}$ is found at this second callsite.

4.4.3. Stage 3: Solving constraints. The guarded constraint propagation is performed on the constraint graph of a procedure P using the rules given in Table I as follows [50] (by considering the new points-to relations added to P's alias graph and the new ones introduced at P's callsites). First, points-to cycles are detected and collapsed to make the constraint graph acyclic. Second, the points-to information is propagated across the existing copy edges. Third, all load and store edges are

DOI: 10.1002/spe

processed, resulting in new copy edges to be added to the constraint graph. The process terminates once a fixed point is reached.

Finally, the interprocedural points-to side effects of P are simply expressed on its virtual variables with one single transfer function, denoted $Trans_P$. In particular, $Trans_P(V_i^f)$ is directly read off as the points-to set of V_i^f in P's constraint graph. However, V_{i+1}^f is not included because $*V_i^f = V_{i+1}^f$, that is, $\overbrace{*\cdots*}\, f = \overbrace{*\cdots*}\, f$ with $i+1$ $*$'s in both sides represents a no-op (assignment).

Example 6
Consider Figure 5(c). The points-to set at a node is not shown completely to avoid cluttering. After a fixed point has been reached, the points-to sets $\{(true, \mathtt{g})\}$ and $\{(true, V_2^{\mathtt{x}}), (\mathbb{C}_{1,1}^{\mathtt{x},\mathtt{y}}, \mathtt{g}\}$ will be directly available at $V_1^{\mathtt{y}}$ and $V_1^{\mathtt{z}}$, respectively. So, the transfer function $Trans_{\mathtt{foo}}$ as shown is trivially available.

4.5. Soundness and context sensitivity

As ICON represents a generalisation of Andersen's analysis with context sensitivity being realised. It suffices to argue briefly why the soundness of our analysis is still maintained. An analysis is *sound* for a program if it over-approximates its points-to relations. In addition, we also examine the precision achieved by ICON in terms of the accuracy of its points-to information.

Theorem 1 (Soundness)
ICON is sound.

Proof
Andersen's analysis is sound. It suffices to show that its soundness is still maintained by ICON in achieving context sensitivity in its three phases in Algorithm 1. Pre-analysis is sound as Steensgaard's analysis is. Our top-down analysis performs essentially Andersen's analysis with the side effects of all callsites ignored. Such side effects are considered when performing Andersen's analysis during our bottom-up analysis. Finally, all aliasing relations at procedure entries are tracked by Lemmas 1 and 2, and our guarded constraint resolution rules are the same as the standard ones except that copy edges are guarded by aliasing relations correctly established by Lemma 2. □

Recall that when the points-to graph G_f for a formal parameter f of a procedure P contains a cycle at its end, $V^f_{\mathsf{depth}(f)}$ stands for all $\overbrace{*\cdots*}\, f$'s with $\mathsf{depth}(f)$ or more $*$'s in P. In this case, some points-to relations may be over-approximated when P is analysed. Therefore, the following theorem is stated under a caveat related to such imprecision inherited from Steensgaard's analysis.

Theorem 2 (Context sensitivity)
Suppose the points-to graph G_f obtained in pre-analysis is acyclic for every formal parameter f of every procedure P in a program. Then, ICON is context sensitive for the program (1) by at least 1-callsite and (2) by (acyclic) call paths if the formal parameters of every procedure are alias free.

Proof
If G_f is acyclic, then every virtual variable V_i^f for every procedure P represents precisely the set of non-local locations that may be passed from its callers, that is, $\overbrace{*\cdots*}\, a$ with exactly i $*$'s for each corresponding actual parameter a. Therefore, statement (1) is true because ICON's aliasing information is at least 1-callsite accurate (line 6 in Algorithm 4). In the absence of aliasing at all procedure entries, the side effects of every procedure are accounted for as if the procedure were cloned at each of its callsites. Thus, statement (2) is true. □

It is possible to capture better the aliasing information at a procedure entry by using a more precise but more expensive pre-analysis or building its virtual variables on the fly together with ICON.

DOI: 10.1002/spe

```
void goo() {
  int i, j;
  int *p, *q = &i
  int *a, *b = &j;
  bar(&p, q);
  bar(&a, b);
}
void bar(int **x, int *y) {
  foo(x, y);
}
void foo(int **u, int *v) {
  *u = v;
}
```

(a)

```
int g;
void goo1() {
  int *a, *b, *x;
  x = bar(&a, &b);
}
void goo2() {
  int *c, *y;
  y = bar(&c, &c);
}
int* bar(int **x, int **y) {
  int *z;
  foo(x, y, &z);
  return z;
}
void foo(int **u, int **v, int **w) {
  *u = &g;
  *w = *v;
}
```

(b)

Figure 6. Context sensitivity of ICON.

This does not appear to be necessary in practice as Steensgaard's analysis is a good choice for our pre-analysis. Precision loss is small when some points-to graphs G_f's have cycles at their tails (Section 4.2). For the 16 SPEC benchmarks used, `gcc` is the worst but with only 3.5% of all G_f's containing cycles. The average across these benchmarks is only 1.3%.

Example 7

Let us illustrate Theorem 2 with two small programs. ICON is context sensitive by call paths in Figure 6(a) due to the absence of aliases. Propagating the modification side effects of `foo` with $Trans_{\texttt{foo}}(V_1^{\texttt{u}}) = \{(true, V_1^{\texttt{v}})\}$ into `bar`, we obtain $Trans_{\texttt{bar}}(V_1^{\texttt{x}}) = \{(true, V_1^{\texttt{y}})\}$. By propagating the modification side effects of `bar` into its two callsites, we obtain $\texttt{p} \xrightarrow{true} \texttt{i}$ and $\texttt{a} \xrightarrow{true} \texttt{j}$ context sensitively. In Figure 6(b), which is slightly modified from Figure 1(b), ICON is only 1-callsite context sensitive. Given that $V_1^{\texttt{x}} = \{\texttt{a}, \texttt{c}\}$ and $V_1^{\texttt{y}} = \{\texttt{b}, \texttt{c}\}$, $V_1^{\texttt{z}} \xrightarrow{true} \texttt{g}$ is created when the callsite to `foo` in `bar` is analysed. As a result, $x \xrightarrow{true} \texttt{g}$ is generated when the callsite to `bar` in `goo1` is analysed. Similarly, $y \xrightarrow{true} \texttt{g}$ is generated when the callsite to `bar` in `goo2` is analysed. However, the spurious $x \xrightarrow{true} \texttt{g}$ will be avoided if a 2-callsite-context-sensitive analysis is used.

By using virtual variables, ICON is more precise than a 1-callsite-sensitive analysis (in the same inclusion-based framework with recursion cycles collapsed in both cases) and makes a good tradeoff between efficiency and precision as evaluated extensively in the following text.

5. EVALUATION

In our experimental validation, we show that ICON has met the three design principles mentioned earlier in Section 1: precision, efficiency and simplicity. We present our results and analysis for 21 C/C++ programs with a total of 2.7 MLOC, including the 15 C programs and 1 C++ from SPEC2000 (600 KLOC) as well as five open-source C applications (2.1 MLOC). The five applications are `wine-0.9.24` (a tool that allows windows applications to run on Linux), `icecast-2.3.1` (a steaming media server), `gdb-6.8` (a debugger), `httpd-2.0.64` (an HTTP server) and `sendmail-8.14.2` (a general-purpose Internet email server).

Some statistics on these 21 C/C++ programs, which are obtained on their intermediate representations (including library code) before procedure pointers are resolved, are given in Table II. We will

DOI: 10.1002/spe

Table II. Benchmark characteristics (the last two columns are produced with no procedure pointers resolved).

Program	KLOC	#Procs	#Pointers	#Loads+Stores	#Callsites		#SCCs	Largest SCC
					Total	Indirect		
164.gzip	8.6	113	3004	586	418	2	0	0
175.vpr	17.8	275	7930	2160	1995	2	0	0
176.gcc	230.4	2256	134380	51543	22353	140	179	398
177.mesa	61.3	1109	44582	17320	3611	671	1	1
179.art	1.2	29	600	103	163	0	0	0
181.mcf	2.5	29	1317	526	82	0	1	1
183.equake	1.5	30	1203	408	215	0	0	0
186.crafty	21.2	112	11883	3307	4046	0	5	2
188.ammp	13.4	182	9829	1636	1201	24	6	2
197.parser	11.4	327	8228	2597	1782	0	42	3
252.eon	41.2	1296	41950	4001	12733	80	17	1
253.perlbmk	87.1	1079	54816	20900	8470	58	12	322
254.gap	71.5	857	61435	22840	5980	1275	33	20
255.vortex	67.3	926	40260	11256	8522	15	12	38
256.bzip2	4.7	77	1672	434	402	0	0	0
300.twolf	20.5	194	20773	8657	2074	0	5	1
gdb-6.8	474.5	7810	337706	105917	52 462	2967	170	128
httpd-2.064	128.1	3000	60027	18450	3959	200	12	4
icecast-2.3.1	22.3	603	15098	9779	877	40	14	1
sendmail-8.14.2	115.2	2656	107242	29220	16973	381	31	76
wine-0.9.24	1338.1	77829	1330840	137409	362787	23523	251	313

also briefly discuss and analyse our results on the 12 C/C++ benchmarks from SPEC2006, which reveal the same trend as the 21 programs focused on in this paper.

Our computer platform is a 3.0 GHz quad-core Intel Xeon running Red Hat Enterprise Linux 5 (Linux kernel version 2.6.18) with 16 GB memory.

5.1. Methodology

There are only a few earlier attempts [42–45] to achieve context sensitivity on top of Andersen's analysis for C/C++. As discussed in Section 6, FULCRA [43] represents a state-of-the-art solution. It is also the most precise inclusion-based analysis because it clones (conceptually) the statements in a procedure with interprocedural points-to side effects and inlines them at its callers.

Therefore, we compare ICON and FULCRA in terms of their efficiency and precision on analysing large C/C++ programs. Just like ICON, FULCRA also collapses the recursion cycles (or SCCs) in a program so that the procedures in an SCC are analysed context insensitively. Otherwise, FULCRA is even more unscalable, especially for large programs [43].

We show that ICON spends just under 35 min on analysing all the 21 programs in the benchmark suite (an average of less than 1.7 min per program) while achieving nearly the same precision as that of FULCRA. In contrast, FULCRA spends over 2 h in analysing 19 programs and fails to run to completion in 5 h for the remaining two. To highlight further the importance of our parameterisation-based approach, we also demonstrate the performance advantages of ICON over NONPA. Recall that NONPA is a non-parameterised version of ICON. So, these two analyses have exactly the same precision.

We measure the precision of an analysis in terms of its capability in alias disambiguation and the quality of the SSA form constructed for a program. These two metrics are believed to be critically important in determining the effectiveness of compiler optimisations.

5.2. Implementation

We have implemented ICON, NONPA and FULCRA in Open64 (version 5.0), an open-source industrial-strength compiler, on its High WHIRL IR at IPA (interprocedural analysis) phase. All

DOI: 10.1002/spe

the analyses are offset-based field-sensitive, by using an existing field-sensitive analysis module in Open64 modified to work with our guarded constraint resolution. In this module, the positive weight cycles that arise from processing fields of struct objects [49] are detected and collapsed. The maximum number of offsets considered for a struct is 256, with the last one representing the 256th and all subsequent offsets available in the struct. However, arrays are considered monolithic. Heap objects are modelled with context-sensitive heap cloning [51, 52] for allocation wrappers. All wrappers are identified and treated as allocation sites in the manner as described in [53, 54]. All the three analyses are compiled under the optimisation flag '-O2' in Open64.

We have implemented FULCRA by following its algorithms [43]. For its top-down phase, we use the code implemented by the Open64 team and already made available in Open64, which is implemented by following the rules in [43, Figure 4.3]. For its bottom-up phase, we coded its summarisation using [43, Figure 4.11] and constraint compaction using [43, Figure 4.15].

We have extended an existing implementation of wave propagation [50] in Open64 to support constraint resolution for all the three analyses evaluated. As discussed in Section 2.1, global variables are tracked separately for efficiency considerations, without participating in procedure summarisation. It is worth mentioning again that in each analysis, the recursion cycles (or SCCs) in a program are merged so that the procedures in each SCC are analysed context insensitively.

5.3. Results and analysis

We first show that ICON is nearly as precise as FULCRA (which is context sensitive by acyclic call paths) for the 21 programs given in Table II. Note that NONPA is a non-parameterised version of ICON and thus has the same precision as that of ICON. We then analyse why ICON achieves such precision even though running significantly faster than FULCRA and NONPA.

5.3.1. Precision. The aliasing information is used extensively to guide compiler optimisations in various passes of Open64's backend, for example, Whirl Optimiser, Loop-Nest Optimiser and Code Generator. As shown in Table III, ICON detects the same or nearly the same percentage of non-alias pairs in all the 21 benchmarks. This percentage metric is used because different aliasing relations

Table III. Alias disambiguation and SSA quality of ICON and FULCRA ('=' means 'same as left').

	Alias disambiguation				SSA quality			
	#Queries		#Not Aliased (%)		#μ's		#χ's	
Program	FULCRA	ICON	FULCRA	ICON	FULCRA	ICON	FULCRA	ICON
164.gzip	1483	=	38.57	=	10582	=	12252	=
175.vpr	23557	=	75.24	=	26386	=	34172	=
176.gcc	101187	101134	31.58	31.11	482184	482296	580224	580263
177.mesa	144328	144398	32.13	32.08	30536	30541	71923	71935
179.art	3772	=	84.51	=	2697	=	3396	=
181.mcf	8588	=	30.34	=	1363	=	2625	=
183.equake	9422	=	80.11	80.09	7768	=	8508	=
186.crafty	15545	=	50.12	=	303532	=	243282	=
188.ammp	38893	=	43.87	=	20539	=	28320	=
197.parser	9529	9636	8.36	8.32	23855	=	33161	33165
252.eon	84838	=	18.56	=	196940	=	309167	=
253.perlbmk	98090	=	12.81	=	135050	135059	175081	175097
254.gap	19360	=	2.11	=	68632	=	105058	=
255.vortex	81542	=	32.31	=	210699	=	302539	=
256.bzip2	2404	=	54.91	=	4174	=	5515	=
300.twolf	92917	=	70.56	=	60575	=	66026	66158
gdb-6.8	**>5 h**	540650	**>5 h**	17.07	**>5 h**	722775	**>5 h**	959068
httpd-2.064	56689	=	11.41	=	167146	=	184687	=
icecast-2.3.1	7774	=	25.94	=	18810	=	32910	=
sendmail-8.14.2	144336	144398	30.71	30.67	517244	517286	581516	581599
wine-0.9.24	**>5 h**	434805	**>5 h**	29.88	**>5 h**	1743726	**>5 h**	1865159

DOI: 10.1002/spe

detected earlier can affect queries issued later. For both algorithms, the same alias analysis interface, *AliasAnalyzer*, provided in Open64 is used to issue alias queries generated by various compiler optimisations. Only the ones that require the results of a pointer analysis to disambiguate are issued.

Many program analysis and compiler optimisations are nowadays performed on SSA. In Open64, SSA construction is intraprocedural based on the approach introduced in [4]. For each store $*x = y$, a $v = \chi(v)$ operation is introduced for each location v pointed by x. Similarly, for each load $x = *y$, a $\mu(v)$ operation is introduced for each location v pointed by y. When converted to SSA form, each $v = \chi(v)$ is treated as both a def and use of v and each $\mu(v)$ as a use of v. In the absence of strong updates, the def v must incorporate the pointer information from both y and the use v. As shown in Table III, ICON and FULCRA give rise to nearly the same SSA representations in all benchmarks (measured in terms of χ and μ operations). For a benchmark, ICON results in the same SSA as FULCRA if ICON scores two =’s, one for column ‘#μ’s’ and one for column ‘#χ’s’.

These results show that ICON is nearly as precise as FULCRA in terms of the quality of the built SSA form and the precision of the discovered alias information.

5.3.2. Efficiency. From the analysis times given in Table IV for the three analyses compared, we see that ICON is the only one that scales to millions of lines of code.

Comparing ICON *and* FULCRA. FULCRA spends over 2 h analysing the 16 programs from SPEC2000 (totalling 600 KLOC) and fails to terminate within 5 h when analysing `gdb` and `wine` (totalling 1.8 MLOC). In contrast, ICON spends just under 7 min on SPEC2000 and just under 26 min on both `gdb` and `wine` together. For the other three applications, `httpd`, `icecast` and `sendmail`, ICON is also faster.

FULCRA is not scalable when analysing programs with large recursion cycles, as also reported by its author in [43, Table 4.3]. FULCRA conservatively identifies the side-effect-causing constraints (called *critical edges*) of a callee, then ‘inlines’ them at its callers, and finally, resolves them at those callers to achieve cloning-based context-sensitivity. There are three reasons affecting its scalability. First, such critical edges are recomputed (conservatively) whenever a callee is re-summarised. Second, more edges than necessary may be pasted from the callee into a caller, causing a rippling effect upwards its call chains. Third, the pasted edges are solved repeatedly at different callsites. As a

Table IV. Analysis times of ICON, NONPA and FULCRA.

	Analysis time (s)				
Program	ICON	NONPA	ICON Speedup	FULCRA	ICON Speedup
164.gzip	0.03	0.03	1.00	0.03	1.00
175.vpr	0.07	0.16	2.29	0.15	2.14
176.gcc	161.60	689.42	4.27	3805.01	23.55
177.mesa	39.56	87.60	2.21	94.32	2.38
179.art	0.00	0.00	1.00	0.00	1.00
181.mcf	0.02	0.01	0.50	0.01	0.50
183.equake	0.01	0.02	2.00	0.01	1.00
186.crafty	0.08	0.14	1.75	0.13	1.63
188.ammp	0.12	0.20	1.67	0.14	1.17
197.parser	0.44	0.67	1.52	1.33	3.02
252.eon	16.70	93.70	5.61	88.60	5.31
253.perlbmk	81.75	615.32	7.53	3107.28	38.01
254.gap	63.35	290.61	4.59	390.87	6.17
255.vortex	19.59	37.66	1.92	44.10	2.25
256.bzip2	0.01	0.02	2.00	0.02	2.00
300.twolf	0.11	0.41	3.73	0.36	3.27
gdb-6.8	586.51	2958.02	5.04	**>5 h**	> 30
httpd-2.064	67.84	153.22	2.26	148.46	2.19
icecast-2.3.1	7.23	26.56	3.67	27.17	3.76
sendmail-8.14.2	49.75	220.08	4.42	380.08	7.64
wine-0.9.24	948.75	**>5 h**	> 17	**>5 h**	> 17

DOI: 10.1002/spe

Table V. Alias graph Statistics.

Program	Average #nodes in alias graphs over Total #nodes in all constraint graphs	#Aliasing variables per procedure	
		Avg	Max
164.gzip	4.94	0.15	2
175.vpr	13.81	0.71	8
176.gcc	4.95	0.69	12
177.mesa	14.39	0.68	9
179.art	3.33	0.00	0
181.mcf	10.10	1.00	5
183.equake	5.07	0.10	1
186.crafty	1.46	0.05	1
188.ammp	4.41	0.30	3
197.parser	5.72	0.41	6
252.eon	5.27	0.15	4
253.perlbmk	1.78	0.13	5
254.gap	2.59	0.51	8
255.vortex	13.01	0.51	8
256.bzip2	4.25	0.00	0
300.twolf	1.83	0.49	5
gdb-6.8	9.88	0.45	16
httpd-2.064	3.63	0.08	5
icecast-2.3.1	2.20	0.05	2
sendmail-8.14.2	5.98	0.45	12
wine-0.9.24	4.21	0.66	15
Average	5.85	0.36	6

result, many edges in a program's constraint graph are introduced unnecessarily, slowing down the analysis performance. Among the 19 benchmarks analysable by FULCRA, `gcc` is the most costly to analyse, taking a little over an hour to complete, because of a large number of constraints moved from callees to their callers during its analysis. In contrast, ICON spends less than 3 min in analysing this benchmark. It should be noted that both FULCRA and ICON analyse all the 21 programs with their SCCs collapsed.

Comparing ICON *and* NONPA. NONPA does not terminate within 5 h when analysing `wine`. For the remaining 20 benchmarks, NONPA spends 5173.85 s (>86 min) while ICON spends 1094.77 s (<19 min).

Full parameterisation has no benefits for small programs such as `art`, `equake` and `gzip` or even hurts performance in the case of `mcf` due to the use of virtual variables. However, significant performance improvements occur for large programs such as `eon`, `gap`, `gcc`, `perlbmk`, `gdb`, `sendmail` and `wine`. This is because these programs have a larger number of pointers as well as loads and stores (Table II). For some large programs such as `mesa`, `vortex` and `vpr`, the improvements are less remarkable because they require more virtual variables to be used, as shown in Table V. In the case of `httpd`, with many pointers as well as loads and stores but few virtual variables, the improvement is small because the number of levels of indirections among its pointers is small (Figure 2).

More analysis. To the best of our knowledge, ICON is the fastest context-sensitive inclusion-based pointer analysis ever reported, at least measured in terms of the 16 C/C++ SPEC2000 benchmarks, five open-source applications, and the 12 C/C++ SPEC2006 benchmarks (discussed in Section 5.4). There are several reasons behind this.

- By using virtual variables to parameterise pointer information, ICON propagates points-to information across copy edges significantly less frequently than FULCRA and NONPA, as shown in Figure 7. This is particularly pronounced in the benchmarks for which ICON achieves the best speedups. The analysis overhead thus incurred is small because the alias graphs, as shown in Table V, are small.

DOI: 10.1002/spe

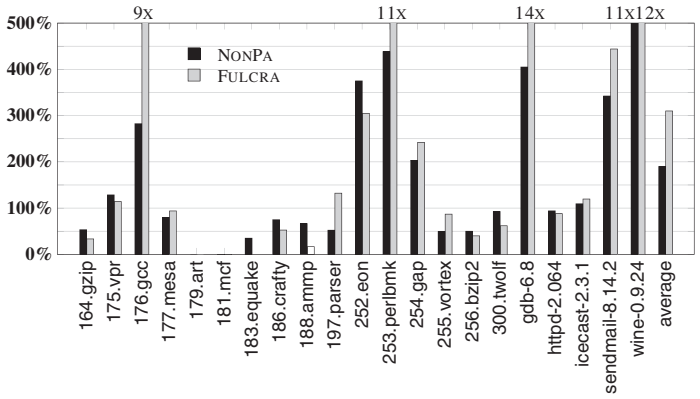


Figure 7. Number of times on processing copy edges by NONPA and FULCRA (normalised w.r.t. ICON).

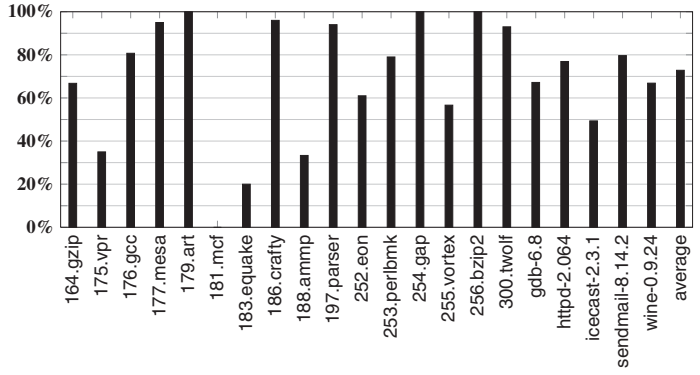


Figure 8. Percentage of virtual variables discovered after the first top-down phase is completed.

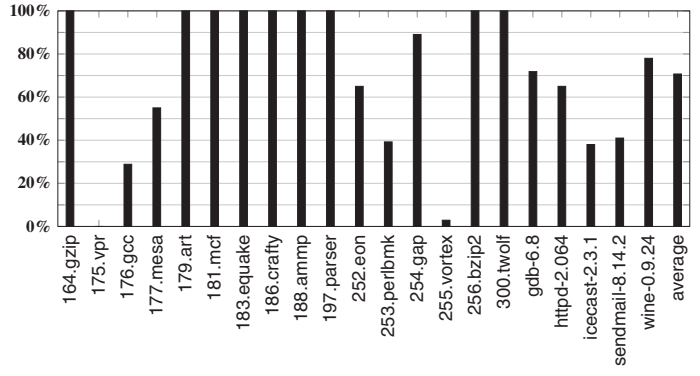


Figure 9. Percentage of procedure pointer targets resolved after the first top-down phase is completed.

- Analyzing a program top-down rather than bottom-up first has two benefits. After the first top-down phase is completed, two observations are made. First, the majority of aliasing variables (77.88% on average) in a program have been introduced as shown in Figure 8. In several small to medium benchmarks, such as `ammp`, `equake`, `mcf` and `vpr`, however, most aliases still need to be discovered. Second, the majority of procedure pointer targets in a program have also been resolved, as shown in Figure 9. However, in some benchmarks, such as `gcc` and `vortex`, most of their indirect call edges will have to be resolved later, because they need to be discovered with at least one bottom-up phase being performed.

DOI: 10.1002/spe

- Pre-analysis helps avoid unnecessary checks for aliasing (lines 3 and 4 in UPDATEALIAS-GRAPH). Most of PFPs are not aliased as shown in Figure 10. On average, around 85% of the procedures in a program are alias free.

Memory usage. Table VI compares ICON, FULCRA and NONPA in terms of memory usage. For the 20 benchmarks that can be analysed by NONPA, ICON consumes 16% more memory on average. ICON needs extra space to store virtual variables but saves space as parameterisation reduces the number of copy edges created (Figure 7). Compared with NONPA, ICON uses more memory in 15 of the 20 benchmarks that are analysable by NONPA. For some small benchmarks with a few procedures and pointers (Table II), such as `art`, `mcf` and `equake`, ICON consumes slightly more memory than NONPA. Some relatively large pointer-intensive benchmarks (Figure 2), including `mesa`, `eon`, `gap`, `vortex` and `httpd`, require at least 20 MB of memory each to analyse in either case. For these benchmarks, ICON requires more space to store virtual variables and thus consumes slightly more memory than NONPA. For the largest benchmarks, such as `gcc`, `perlbmk` and `gdb`, ICON has succeeded in exploiting parameterisation to reduce significantly the number of copy edges

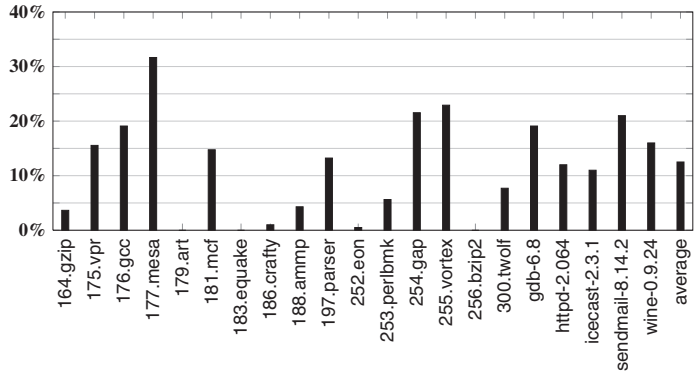


Figure 10. Percentage found to be aliased among all pairs of pointer formal parameters by pre-analysis.

Table VI. Memory usage of ICON, NONPA and FULCRA.

	Memory usage (MB)				
Program	ICON	NONPA	ICON increase (ICON/NONPA)	FULCRA	ICON increase (ICON/NONPA)
164.gzip	2.90	1.15	2.52	1.23	2.36
175.vpr	9.40	8.23	1.14	8.15	1.15
176.gcc	425.86	535.11	0.80	754.38	0.56
177.mesa	67.36	60.48	1.11	87.37	0.77
179.art	0.83	0.82	1.01	0.82	1.01
181.mcf	1.09	1.09	1.00	1.06	1.03
183.equake	1.07	1.07	1.00	1.07	1.00
186.crafty	5.93	5.32	1.11	5.41	1.10
188.ammp	7.21	7.01	1.03	6.81	1.06
197.parser	10.85	9.55	1.14	8.56	1.27
252.eon	68.07	58.07	1.17	59.90	1.14
253.perlbmk	131.54	186.18	0.71	306.18	0.43
254.gap	85.85	51.84	1.66	57.19	1.50
255.vortex	38.20	34.88	1.10	35.96	1.06
256.bzip2	2.18	1.67	1.31	1.71	1.27
300.twolf	12.22	12.13	1.01	12.21	1.00
gdb-6.8	1633.45	2333.45	0.70	**>5 h**	**>5 h**
httpd-2.064	29.99	20.09	1.49	23.14	1.30
icecast-2.3.1	20.19	16.87	1.20	16.66	1.21
sendmail-8.14.2	132.58	116.11	1.14	120.38	1.10
wine-0.9.24	1820.88	**>5 h**	**>5 h**	**>5 h**	**>5 h**

DOI: 10.1002/spe

created by NONPA (Figure 7). As a result, for `gcc`, `perlbmk` and `gdb`, ICON consumes only 80%, 71% and 70%, respectively, of the amount of memory consumed by NONPA.

Let us now compare ICON and FULCRA. For the 19 benchmarks that can be analysed by FULCRA, ICON consumes 12% more memory on average. The extra amount of memory incurred by ICON for small and medium benchmarks is negligible. However, for large benchmarks, ICON uses less memory than FULCRA. In the case of `gcc`, `mesa` and `perlbmk`, ICON consumes only 56%, 77% and 43%, respectively, of the amount of memory consumed by FULCRA, as FULCRA introduces a large number of extra constraints when promoting side-effect-causing statements from a callee to its callers, especially in the presence of large recursion cycles.

5.3.3. Simplicity. For this design goal, we aim to leverage the recent advances in inclusion-based analysis so that context sensitivity can be achieved in the same constraint resolution framework. In other words, the existing code base should be reused as much as possible.

ICON's pre-analysis is bootstrapped by the standard Steensgaard's unification-based analysis [2]. During each iteration, both the top-down and bottom-up phases are performed using the same constraint resolution engine based on an existing module for wave propagation [50] in Open64. This module is modified so that guarded copy edges are handled in the presence of aliasing variables.

5.4. SPEC 2006

We briefly discuss the results obtained on comparing ICON and FULCRA using the 12 C programs in SPEC2006 (totalling 1.0 MLOC). ICON is once again nearly as precise as FULCRA in terms of their capabilities in alias disambiguation and the quality of the SSA form built for a program.

ICON spends only less than 14 min (803.04 s) in analysing all the 12 programs in SPEC2006. In contrast, FULCRA spends over 48 min (2922.48 s) in analysing 10 benchmarks and fails to terminate within 5 h for the remaining two benchmarks, `400.perlbench` and `403.gcc`. These are the top two in SPEC2006 ranked in terms of how many pointers and SCCs they contain: `400.perlbench` has 13 283 pointers, 46 792 loads and stores and 30 SCCs (with the largest SCC containing 461 procedures), and `403.gcc` has 35 697 pointers, 123 389 loads and stores and 317 SCCs (with the largest SCC containing 436 procedures). As in the case of Table II, the statistics on SCCs are collected before procedure pointers are resolved. For these two benchmarks, FULCRA is not scalable for the same reason explained earlier for `176.gcc` in Section 5.3.2.

6. RELATED WORK

Context sensitivity. There are many pointer analyses in the literature with different types of flow sensitivity assumed: some are flow sensitive [32–35, 38–40, 46, 55], some are inclusion based [28, 44, 45] and some are unification based [47, 51, 56–58]. To account for the interprocedural side effects of a procedure context sensitively, some pointer analyses resort to cloning [26, 45, 46], while others rely on procedure summarisation [32, 34, 39, 40, 59, 60]. While binary decision diagrams can be used to handle efficiently the exponential number of contexts by exploiting their similarities [26, 60, 61], cloning-based algorithms are still not scalable to large programs. When context sensitivity is considered, different types of precision are distinguished if calling contexts are identified by full call paths [34, 40], assumed aliases at callsites [32, 35], acyclic call paths (with the SCCs in the call graph being collapsed) [34, 40] and approximated call paths within the SCCs [26, 39, 46]. The research described in [29, 51] focuses on achieving scalability for context-sensitive heap cloning and is thus orthogonal to this research.

There are some earlier attempts [42–45] on adding context sensitivity to Andersen's analysis to analyse C/C++ programs. Cloning [45] achieves context sensitivity trivially by analysing different calls to a procedure using different clones of the procedure. FULCRA [43, 44], which improves [42], clones only the side-effect-causing constraints, that is, the so-called critical edges in a procedure. The analysis introduced in [62] is demand driven rather than a whole-program analysis.

DOI: 10.1002/spe

ICON is more powerful than a 1-callsite-sensitive analysis and less so than a cloning-based context-sensitive analysis when recursion cycles are collapsed in all cases (Theorem 2). Therefore, a good balance between efficiency and precision is maintained to make ICON practical for compilers.

Parameterisation. To distinguish the side effects of a procedure context sensitively at its different callsites, symbolic names have been used to represent points-to relations passed from its callers into the procedure, such as invisible blocks [35, 46], auxiliary parameters [47], extended parameters [39] and semi-parameterised spaces [40]. Some important benefits include improved precision (due to more strong updates enabled) if flow sensitivity is considered [39, 40] and faster analysis (by propagating one symbolic name instead of all individual locations abstracted). In [47], its parameterisation is performed on a unification-based analysis. In [39, 46], when flow sensitivity is considered, some aliased symbolic names of a procedure are either allowed or merged, trading precision for efficiency. In [27], the analysis proposed for Java uses parameterised points-to escape graphs to identify memory blocks escaping from a method.

In this paper, we consider inclusion-based analysis without strong updates. The points-to values passed into a procedure are fully parameterised in terms of (symbolic) virtual variables. To the best of our knowledge, this is the first paper exploiting fully parameterised pointer information on inclusion-based pointer analysis, resulting in (1) faster convergence facilitated by parameterised side-effect summarisation and (2) call-path-based context sensitivity in the special case when the formal parameters of every procedure are alias free (Theorem 2).

Side-effect summarisation. There are two earlier summary-based approaches to achieving context sensitivity, by using relevant context inference (RCI) [32] and partial transfer functions (PTFs) [39], which were both originally proposed for flow-sensitive analysis. In the case of inclusion-based analysis, both are not as efficient as ICON. RCI builds one transfer function for a procedure eagerly by assuming the presence of all possible aliases at its entry, which is unnecessarily costly as revealed by (lack of) the aliasing relations in real code given in Table V. On the other hand, building multiple PTFs for a procedure lazily based on different aliasing combinations at its callsites is also unnecessarily costly, as this would require multiple constraint graphs to be constructed for the procedure. By exploiting the absence of strong updates, ICON is designed to perform its analysis quickly by summarising a procedure iteratively in-place, that is, in the same constraint graph of the procedure.

Pre-analysis. An analysis can be bootstrapped with the results of a prior analysis that is faster but less precise [33, 34]. In [40], Steensgaard's analysis is performed first to assign a level to each variable. Then, the program is re-analysed level by level for greater precision. In [54], Andersen's analysis is used to accelerate static detection of memory leaks for C/C++ programs. In this paper, ICON is boosted by must-not-aliased information to accelerate its convergence.

7. CONCLUSION

We introduce a new context-sensitive pointer analysis on top of Andersen's analysis that can scale to millions of lines of C/C++ code, making it deployable in modern optimising compilers to drive advanced compiler optimisations. We have validated its scalability in Open64 using a total of 21 C/C++ programs (totalling 2.7 MLOC), by comparing it with the state of the art. By summarising the points-to side effects of a procedure parametrically using virtual variables, our analysis is significantly faster than the state of the art while yielding nearly the same precision.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and suggestions to the earlier versions of this paper This research is supported by the Australian Research Council (ARC) grants (DP0987236 and DP130101970) and an International Science and Technology Cooperation Program of China (2011DFG13000).

DOI: 10.1002/spe

REFERENCES

1. Andersen LO. Program analysis and specialization for the C programming language. *Ph.D. Thesis*, University of Copenhagen, 1994. (DIKU report 94/19).
2. Steensgaard B. Points-to analysis in almost linear time. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM: New York, NY, USA, 1996; 32–41.
3. Hardekopf B, Lin C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2007; 290–299.
4. Chow F, Chan S, Liu S, Lo R, Streich M. Effective representation of aliases and indirect memory operations in SSA form. *International Conference on Compiler Construction*, Springer, 1996; 253–267.
5. Chaitin GJ. Register allocation & spilling via graph coloring. *SIGPLAN'82: Proceedings of the Sigplan Symposium on Compiler Construction*, ACM: New York, NY, USA, 1982; 98–101.
6. Rau BR. Iterative modulo scheduling: an algorithm for software pipelining loops. *International Symposium on Microarchitecture*, New York, NY, USA, 1994; 63–74.
7. Cai Q, Xue J. Optimal and efficient speculation-based partial redundancy elimination. *International Symposium on Code Generation and Optimization*, IEEE/ACM, 2003; 91–104.
8. Knoop J, Rüthing O, Steffen B. Lazy code motion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '92. ACM: New York, NY, USA, 1992; 224–234.
9. Xue J, Knoop J. A fresh look at partial redundancy elimination as a maximum flow problem. *International Conference on Compiler Construction,* Springer 2006; **26**(2).
10. Li L, Feng H, Xue J. Compiler-directed scratchpad memory management via graph coloring. *ACM Transactions on Architecture and Code Optimization* 2009; **6**(3).
11. Li L, Gao L, Xue J. Memory coloring: a compiler approach for scratchpad memory management. *International Conference on Parallel Architectures and Compilation Techniques*, IEEE/ACM, 2005; 329–338.
12. Udayakumaran S, Barua R. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03. ACM: New York, NY, USA, 2003; 276–286.
13. Yang X, Wang L, Xue J, Deng Y, Zhang Y. Comparability graph coloring for optimizing utilization of stream register files in stream processors. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM: New York, NY, USA, 2009; 111–120.
14. Gao L, Li L, Xue J, Yew P-C. Seed: a statically greedy and dynamically adaptive approach for speculative loop execution. *IEEE Transactions on Computers* 2013; **62**(5):1004–1016.
15. Quinones CG, Madrile C, Sanchez J, Marcuello P, Gonzalez A, Tullsen DM. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005; 269–279.
16. Vachharajani N, Rangan R, Raman E, Bridges MJ, Ottoni G, August DI. Speculative decoupled software pipelining, International Conference on Parallel Architectures and Compilation Techniques, IEEE/ACM, 2007; 49–59.
17. Du Z-H, Lim C-C, Li X-F, Yang C, Zhao Q, Ngai TF. A cost-driven compilation framework for speculative parallelization of sequential programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2004; 71–81.
18. Lhoták O, Hendren L. Context-sensitive points-to analysis: is it worth it? *International Conference on Compiler Construction*, Springer, 2006; 47–64.
19. Lu Y, Shang L, Xie X, Xue J. An incremental points-to analysis with CFL-reachability. *International Conference on Compiler Construction*, Springer, 2013; 61–81.
20. Nguyen PH, Xue J. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. *Australasian Computer Science Conference*, Australian Computer Society, 2005; 9–18.
21. Shang L, Lu Y, Xue J. Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. *IEEE/ACM International Conference on Automated Software Engineering*, 2012; 270–273.
22. Shang L, Xie X, Xue J. On-demand dynamic summary-based points-to analysis. *International Symposium on Code Generation and Optimization*, IEEE/ACM, 2012; 264–274.
23. Smaragdakis Y, Bravenboer M, Lhoták O. Pick your contexts well: understanding object-sensitivity. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM: New York, NY, USA, 2011; 17–30.
24. Sridharan M, Bodík R. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2006; 387–400.
25. Sun Q, Zhao J, Chen Y. Probabilistic points-to analysis for Java. *International Conference on Compiler Construction*, Springer, 2011; 62–81.
26. Whaley J, Lam MS. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2004; 131–144.
27. Whaley J, Rinard M. Compositional pointer and escape analysis for Java programs. *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM: New York, NY, USA, 1999; 187–206.
28. Xiao X, Zhang C. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. *International Symposium on Software Testing and Analysis*, ACM: New York, NY, USA, 2011; 188–198.

DOI: 10.1002/spe

29. Xu G, Rountev A. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. *International Symposium on Software Testing and Analysis*, ACM: New York, NY, USA, 2008; 225–236.
30. Xue J, Nguyen PH. Completeness analysis for incomplete object-oriented programs. *International Conference on Compiler Construction*, Springer, 2005; 271–286.
31. Xue J, Nguyen PH, Potter J. Interprocedural side-effect analysis for incomplete object-oriented software modules. *Journal of Systems and Software* 2007; **80**(1):92–105.
32. Chatterjee R, Ryder B, Landi W. Relevant context inference. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM: New York, NY, USA, 1999; 146.
33. Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code. *International Symposium on Code Generation and Optimization*, IEEE/ACM, 2011; 289–298.
34. Kahlon V. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2008; 249–259.
35. Landi W, Ryder BG. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 1992, 27; 248.
36. Li L, Cifuentes C, Keynes N. Precise and scalable context-sensitive pointer analysis via value flow graph. *International Symposium on Memory Management*, ACM: New York, NY, USA, 2013; 85–96.
37. Li L, Cifuentes C, Keynes N. Boosting the performance of flow-sensitive points-to analysis using value flow. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM: New York, NY, USA, 2011; 343–353.
38. Sui Y, Ye S, Xue J, Yew P-C. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. *Asian Symposium on Programming Languages and Systems*, Springer: Berlin, Heidelberg, 2011; 155–171.
39. Wilson R, Lam M. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 1995, 30; 12.
40. Yu H, Xue J, Huo W, Feng X, Zhang Z. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. *International Symposium on Code Generation and Optimization*, IEEE/ACM, 2010; 218–229.
41. Acharya M, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems. *International Conference on Software Engineering*, ACM: New York, NY, USA, 2011; 746–755.
42. Foster JS, Fähndrich M, Aiken A. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. *Static Analysis Symposium*, Springer, 2000; 175–198.
43. Nystrom E. FULCRA pointer analysis framework. *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, 2006.
44. Nystrom E, Kim H, Hwu W. Bottom-up and top-down context-sensitive summary-based pointer analysis. *Static Analysis Symposium*, Springer, 2004; 165–180.
45. Nasre R, Govindarajan R. Prioritizing constraint evaluation for efficient points-to analysis. *International Symposium on Code Generation and Optimization*, IEEE/ACM, 2011; 267–276.
46. Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 1994; 242–256.
47. Liang D, Harrold M. Efficient computation of parameterized pointer information for interprocedural analyses. *Static Analysis Symposium*, Springer, 2001; 279–298.
48. Heintze N, Tardieu O. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2001; 263.
49. Pearce D, Kelly P, Hankin C. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2007; **30**(1):4.
50. Pereira F, Berlin D. Wave propagation and deep propagation for pointer analysis. *International Symposium on Code Generation and Optimization*, IEEE/ACM, 2009; 126–135.
51. Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world, ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM: New York, NY, USA, 2007, 42; 289.
52. Sui Y, Li Y, Xue J. Query-directed adaptive heap cloning for optimizing compilers. *International Symposium on Code Generation and Optimization*, ACM/IEEE, 2013; 1–11.
53. Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2007; 480–491.
54. Sui Y, Ye D, Xue J. Static memory leak detection using full-sparse value-flow analysis. *International Symposium on Software Testing and Analysis*, ACM: New York, NY, USA, 2012; 254–264.
55. Hackett B, Aiken A. How is aliasing used in systems software? *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM: New York, NY, USA, 2006; 69–80.
56. Das M, Liblit B, Fähndrich M, Rehof J. Estimating the impact of scalable pointer analysis on optimization. *Static Analysis Symposium*, Springer, 2001; 260–278.
57. Fähndrich M, Rehof J, Das M. Scalable context-sensitive flow analysis using instantiation constraints. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2000, 35; 253–263.
58. Liang D, Harrold M. Efficient points-to analysis for whole-program analysis. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM: New York, NY, USA, 1999; 199–215.

DOI: 10.1002/spe

59. Dillig I, Dillig T, Aiken A, Sagiv M. Precise and compact modular procedure summaries for heap manipulating programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2011; 567–577.
60. Zhu J, Calman S. Symbolic pointer analysis revisited. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004; 157.
61. Berndl M, Lhoták O, Qian F, Hendren L, Umanee N. Points-to analysis using BDDs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2003, 38; 103–114.
62. Zheng X, Rugina R. Demand-driven alias analysis for C. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM: New York, NY, USA, 2008; 197–208.

DOI: 10.1002/spe