Scalable, Precise Context-Sensitive Top-Down Process for Modular Points-to Analysis

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

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

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

ABSTRACT

The main loop of modular points-to analysis typically consists of two major phases. The first, a bottom-up process, *transforms each procedure into a self-contained* module *that encapsulates all the internal dataflow along with side effects from procedure calls. The second, a* top-down process, *provides the input points-to information to each procedure and computes the actual points-to information.*

We propose an ultimately precise (with respect to context-sensitivity), yet very efficient holistic *top-down process that produces points-to information valid across all call paths. When combined with our previously proposed, efficient and exact bottom-up process* [22], *the resultant modular points-to analysis is equivalent to that of a context-sensitive algorithm entirely reliant on exhaustive inlining. The scalability and usefulness of the proposed approach is demonstrated by empirical results.*

1. INTRODUCTION

Modern programming practices encourage code reuse and result in extensive sharing of procedures throughout a program. In the domain of points-to analysis, pervasive use of shared procedures exacerbates the problem of unrealizable interprocedural dataflow [25].

Context-sensitive algorithms deliver a higher degree of precision than their context-insensitive counterparts because these analyses are able to avoid cross-contamination through shared procedure calls. Furthermore, in a context-sensitive algorithm allowing *heap object specialization*, the spurious dataflow through aggregated *static* heap objects can also be alleviated to a certain degree.

The major challenge in context-sensitive points-to analysis lies in regulating complex interprocedural dependencies formed by destructive updates via procedure calls, termed *side effects* in this paper. A straightforward way to solve the problem is by completely *inlining* procedure calls and specializing the accompanying variables, thus reducing the problem to a single intraprocedural one. However, exponential code-size growth makes this naïve approach impractical even for moderate-size programs with acyclic call graphs.

Modularization [7, 8] is one of the popular methods used to implement context sensitivity while avoiding serious code-size growth. At a high level, the main loop of modular points-to analysis consists of two major phases. The first phase is a *bottom-up process* that transforms each procedure into a self-contained *module*. Each module encapsulates all of the procedure's internal dataflow as well as all possible side effects from its descendants in the call graph. By doing so, the interprocedural dataflow problem is reduced into a series of topologically-sorted intraprocedural problems. The most important issue in the bottom-up process is the efficient capture of all potential procedural side effects and then concisely representing those side-effects in a procedure summary. We have proposed one such a method that *compacts* procedure modules into summaries without altering their external behavior [22].

The main focus of this paper is on the second phase which is a *top-down process*. This process provides the input points-to information to each procedure and then computes the actual, or concrete, points-to information for that procedure. Two commonly used approaches to the top-down process that appear in literature are the *(call-)path-specific* approach [7, 15] and the *(call-)path-unspecific* approach. The path-specific approach is effective when points-to information is only needed along a particular call path and the result obtained is very precise. However, this approach does not scale well when points-to information needs to be exhaustive across all call paths (for example, to precompute alias relationships for use by a scheduler) because there may be an exponential number of distinct call paths. In order to scalably generate points-to information valid across all call paths, one must avoid explicit enumeration of points-to information from individual call paths. However, at least in [8, 15], the means used to achieve the path-unspecific result leads to a partial loss in context sensitivity, which also turns into precision degradation.

In this paper, we propose an ultimately precise (with respect to context sensitivity) yet extremely efficient top-down process that produces points-to information representative of all call paths. In particular, its result is equivalent to that collected from exhaustive runs of the path-specific top-down process (one run for each distinct call path). Consequently, when combined with our previously proposed, exact bottom-up process [22], the resultant modular points-to analysis is equivalent to a context-sensitive algorithm entirely reliant on exhaustive inlining yet operates in an incomparably scalable way.

The proposed modular points-to analysis is implemented as part of a flexible points-to analysis framework in the IMPACT C compiler, allowing exploration of the multidimensional points-to analysis design space. Using this framework we performed a number of empirical studies presented in Section 8. These demonstrate that, with few exceptions, context-sensitivity can be achieved with a reasonable cost across large benchmarks. Furthermore, context sensitivity and heap-object specialization are shown, in many cases, to

$$\frac{\alpha \in A}{\alpha} \quad (1)$$

$$\frac{u := v \quad v := \&w}{u := \&w} \quad (2)$$

$$\frac{u := \&v \quad *u := w}{v := w} \quad (3)$$

$$\frac{u := *v \quad v := \&w}{u := w} \quad (4)$$

Figure 1: Small-step executions given an assignment set A. An assignment above a line is the *input* and the one below is the *output* of the small-step execution.

provide a significant precision improvement.

2. INTRAPROCEDURAL SETTING

To simplify our initial investigations, the proposed points-to analysis operates flow-insensitively. Though supported in the implemented algorithm, the issue of field sensitivity will not be discussed until Section 7. Given these two simplifications, the following four kinds of assignments suffice[1]

$$u := \&v \mid u := v \mid *u := v \mid u := *v$$

where u,v are variables, & the address operator, $*$ the dereference operator, whose meanings follow those in the C language.

2.1 Derivation tree

In the literature many different forms of flow-insensitive approaches have been proposed [3, 10, 21, 27]. We have chosen the inclusion-based approach [3], which is the most precise among those, while keeping the cost within a reasonable bound. Given an assignment set A, our abstract execution can be modeled as a *derivation tree* constructed by the small-step executions in Figure 1, presented in the style of [18]. The rule (1) serves as a seed, upon which a derivation tree can be grown by applying other inference rules. For instance, given

`A` = {`a:=&b, b:=&c, k:=a, k:=b, l:=k, p:=l, q:=*p`}

the following derivation tree can be formed, where a label for the inference rule has been placed beside each small-step execution.

$$\dfrac{\dfrac{\dfrac{\texttt{q:=*p} \qquad \dfrac{\dfrac{\texttt{k:=a} \quad \texttt{a:=\&b}}{\texttt{p:=k} \qquad \texttt{k:=\&b}}\,(2)}{\texttt{p:=\&b}}\,(3)}{\texttt{q:=b}} \qquad \texttt{b:=\&c}}{\texttt{q:=\&c}}\,(2)$$

In this case, $A \vdash$ `q:=&c` states that `q:=&c` has been derived from A according to rule (1). By default, an assignment explicitly in A is considered to be derived from A.

[1] In some code examples, we may abuse the syntax for clarity. For instance, `*k:=&l` is understood as `*k:=x` and `x:=&l`, where `x` is a fresh variable appearing nowhere in the program.

2.2 Derivation graph

Given an assignment set A, its points-to information can be collectively computed by an algorithm whose complexity is cubic in the size of A. This will be named the *intraprocedural points-to* (IA) algorithm. The main data structure maintained by the IA algorithm is called a *derivation graph*, which can be understood as an overlay of multiple derivation trees.

Each vertex in the derivation graph corresponds to an assignment derivable from A and each edge to an input-output relationship between assignments formed by small-step executions. In particular, we draw an edge from a vertex n to m if and only if a small-step execution having n as one of its inputs and m as an output can be taken.

A complete derivation graph for the set `A` introduced in Subsection 2.1 is shown in Figure 3. In this figure, we have explicitly introduced a join from two incoming edges to make the effect of small-step executions clearer. The derivation tree shown in Subsection 2.1 is a subgraph in this derivation graph.

3. INTERPROCEDURAL SETTING

Our modular points-to analysis begins with a call graph initially devoid of indirect calls and is updated iteratively throughout the analysis process until no new call targets for indirect calls are found. Since our modular points-to analysis is essentially inline-based, termination can be guaranteed by merging recursive procedures into a single procedure. Without losing generality, the remainder of this paper will only consider programs with a fixed acyclic call graph.

To further simplify the presentation, the assumed language does not contain a return statement; its effect can be replaced by an extra parameter into which the address of the actual return variable is passed. However, in the actual implementation, we have extended the techniques introduced in this paper to work directly with a return statement. For similar reasons, we have excluded global variables from the discussion until Section 7.

3.1 Context-insensitivity

As briefly mentioned in Section 1, the key challenge to implementing context-sensitive points-to analysis lies in dealing with interprocedural dependencies caused by side effects[2]. Consider the following example.

```
P1: aa()       {bb1(&a,&b,&a); bb2(&c,&d,&a);}
    bb(k,l,m)  {*k:=l; n:=*m;}
```

A context-insensitive (CI) algorithm blindly collects all the assignments in the program into a single set along with a number of assignments to mimic the effects of parameter passing between procedure calls, called *param-assignments*. From this set, a single run of the IA algorithm produces points-to information for all the variables in the program. Applying the CI algorithm to the example `P1` yields the three derivation trees shown in Figure 2, among others. Brackets mark param-assignments and the subscripted number indicates the particular call site.

[2] Intuitively speaking, when a callee makes an update visible to its caller, we say that there is a side effect. In the discussion of this paper, it is caused when a left-dereference assignment updates a variable non-local to the procedure.

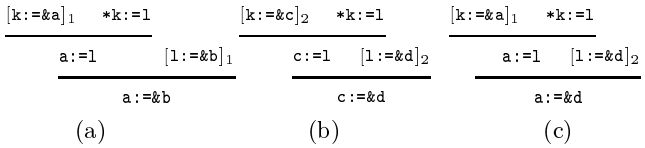


Figure 2: Three possible derivation trees that result from applying the CI algorithm to P1. Among them, only (c) is interprocedurally spurious.

Derivation trees (a) and (b) follow a single call path and produce an intended result. However, the derivation (c) shows unrealizable, or *spurious*, dataflow occurring between two call paths. It does not correspond to any actual execution sequence because `a:=l` is valid only within the call site `1` while `l:=&d` is valid only within the call site `2`.

This problem, termed the *spurious dataflow problem*, is rooted in the existence of procedural side effects. It occurs when interprocedural dataflow formed from two different call paths to the same procedure interact due to a lack of context differentiation.

Even for a program lacking side effects, another kind of precision loss can occur, termed the *context-loss problem*. In this case, the overlapping of interprocedural dataflow does not cause any spurious interactions. However, a lack of context information causes a different problem. Consider the following example.

```
P2: cc()    { dd1(&a,&b); dd2(&b,&a); }
    dd(p,q) { r:=*p; s:=*q; }
```

It is apparent that `dd()` has no side effects and that both `*p` and `*q` can point to `a` and `b`. The direct use of the points-to information will answer positively to the *alias* query between `*p` and `*q`. However, this is a spurious alias result because `p` and `q` never point to the same variable simultaneously (i.e. along any given call path). *The main contribution of this paper, summarized in Section 4, is to completely eliminate both the spurious dataflow and context-loss problems in a scalable way.*

3.2 Inlining approach

A context-sensitive algorithm reliant on exhaustive inlining, sometimes simply referred to as *the* CS algorithm, is an intuitive but unscalable way to resolve both the spurious dataflow and context-loss problems. The inlining transforms the interprocedural problem into a single, potentially huge, intraprocedural problem. Context sensitivity is maintained by specializing the variables of each inlined copy. This specialization is shown using subscripts, as in the following.

```
P1': aa'() {
        k1:=&a; l1:=&b; m1:=&a; *k1:=l1; n1:=*m1;
        k2:=&c; l2:=&d; m2:=&a; *k2:=l2; n2:=*m2;
     }
```

The first line corresponds to the call site `1` in `aa()`. The first three assignments are for parameter passing while the last two are from the body of `bb()`. Note that the local variables of `bb()` have been specialized using `1` as a subscript. The second line corresponds to the call site `2` in a similar way. The spurious dataflow problem has been resolved naturally since variable specialization prevents any interaction between side effects across different call sites. Applying the CS algorithm to the second example results in the following.

```
P2': cc'() {
        p1:=&a; q1:=&b; r1:=*p1; s1:=*q1;
        p2:=&b; q2:=&a; r2:=*p2; s1:=*q2;
     }
```

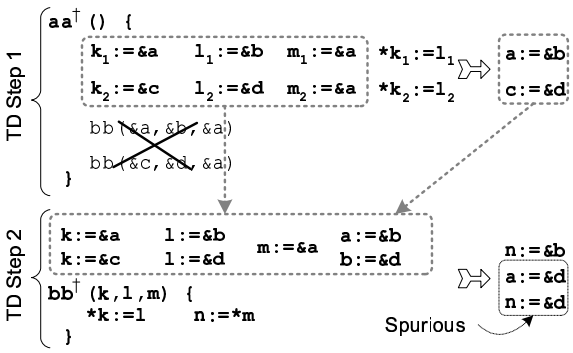
In this form, it is more apparent that `*p` and `*q` are not aliases, since neither the pair (p_1, q_1) nor the pair (p_2, q_2) point to the same variable.

3.3 Modular approach

Modularization [7, 8, 15] provides an alternative mechanism to regulate interprocedural dependencies. As mentioned in Section 1, modular points-to analysis consists of two phases. The bottom-up process explicitly yet *concisely* moves side-effect production from callees to callers, thus cutting the upward interprocedural dependency between them. We have previously proposed one such method [22]. The top-down process then computes the concrete points-to information, by traversing modules in a topological order and transferring points-to information downward. Since we are primarily interested in computing a result valid across all call paths, the following focuses on a path-unspecific approach.

In example `P1` from Section 3.1, `*k:=l` is the only side-effect production in `bb()`. Therefore, to cut the dependency from `aa()` to `bb()`, it is the only assignment that needs to be inlined into `aa()`.

The result of explicitly including these side-effects is shown by the *module* $\texttt{aa}^\dagger\texttt{()}$ below. The consequence of the bottom-up process is that the $\texttt{aa}^\dagger\texttt{()}$ is no longer dependent on $\texttt{bb}^\dagger\texttt{()}$ because it contains specialized replicas of `bb()`'s side effect production. Also note that the replicas of `*k:=l` in $\texttt{aa}^\dagger\texttt{()}$ are no longer side-effects but, instead, are purely local to $\texttt{aa}^\dagger\texttt{()}$.



With the bottom-up process completed, the first step of the top-down process is to derive the concrete points-to information for $\texttt{aa}^\dagger\texttt{()}$. The side-effect specialization performed by the bottom-up process allows the IA algorithm to be applied to $\texttt{aa}^\dagger\texttt{()}$ in isolation. The only concrete points-to results derived are `a:=&b` and `b:=&d`, thus, as expected, no spurious dataflow occurs.

The second step is to derive the concrete points-to information for $\texttt{bb}^\dagger\texttt{()}$. The necessary output-to-input transfer from $\texttt{aa}^\dagger\texttt{()}$ to $\texttt{bb}^\dagger\texttt{()}$ is depicted in the diagram[3].

[3] It corresponds to all the points-to relationships reachable from the parameters of $\texttt{bb}^\dagger\texttt{()}$.

In bb†(), however, the assignment *k:=l combined with k:=&a and l:=&d, again causes the variable a to point to d[4], eventually causing the spurious points-to information n:=&d at the assignment n:=*m.

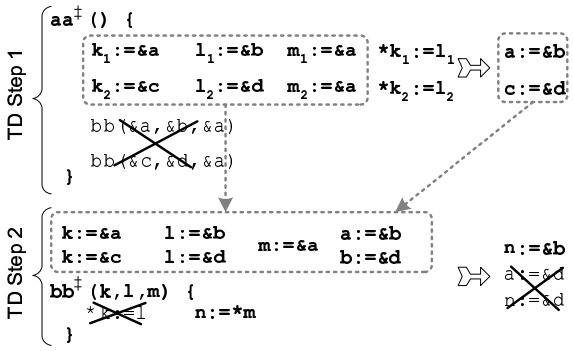
This example shows that in modular points-to analysis, spurious dataflow may corrupt points-to results not only from the perspective of callers, but also from the callees. In addition, since no path-specific information is available from such a path-unspecific top-down process, it is apparent that the context-loss problem, also mentioned Subsection 3.1, may still occur.

4. OVERVIEW

As made apparent in Section 3.3, the spurious dataflow problem can occur during both the bottom-up and top-down processes. The bottom-up process resolves this problem for callers, however it does not aid callees much by itself.

A surprisingly simple remedy for the spurious dataflow seen by callees can be found through the following observation: All callees' side-effects have been replicated and exist locally within callers. Since callers completely rely on their local copies, they no longer depend on their callees. If callees are permitted to consume the consequences of their callers' side-effect copies, callees no longer require the original version. This leaves the original side-effects entirely redundant allowing their erasure. *The* clean-up *of the (now redundant) original side-effects eliminates all sources of spurious dataflow from the program.*

Keeping this in mind, we have revised the modularized version of P1 as follows. The assignment *k:=l, which was replicated into aa‡(), has been erased from bb‡().



It is safe to remove *k:=l if the derived points-to information for both aa‡() and bb‡() will be unaffected. The independence of aa‡(), which is identical to aa†(), has already been demonstrated in Section 3.3.

The module bb‡(), even without (original) *k:=l in its body, consumes the exact consequences, a:=&b, now produced by *k1:=l1 in aa‡(), naturally through the parameter passing mechanism m:=&a and the assignment n:=*m. Now, only the expected points-to information n:=&b is produced.

As demonstrated, to completely fix the spurious dataflow problem, the clean-up step is the only extension needed by the compaction-based bottom-up process. We will refer to the combination of the compaction-based bottom-up process and the clean-up step collectively as the *enhanced* bottom-up process. In fact, after the enhanced bottom-up process, bb‡() *truly* does not affect aa‡(). The following version of P1 is equivalent to the diagram above, thus, also equivalent to the original program P1 (with respect to the CS algorithm).

```
P1*: aa*() {
        k1:=&a; l1:=&b;  m1:=&a; *k1:=l1;
        k2:=&c; l2:=&d;  m2:=&a; *k2:=l2;
        bb*1(k1,l1,m1); bb*2(k2,l2,m2);
     }

     bb*(k,l,m) {
         n:=*m; /* *k:=l; */
     }
```

The enhanced bottom-up process transforms a program into a *side-effect-free* program[5], yet with the same points-to information. Since side-effect productions move up to callers until they becomes purely local in nature (thus, no longer generating side effects), after the clean-up step, programs no longer cause any side-effect productions. Sections 5 and 6 will present more details.

With this transformation, as it will be discussed in the following subsection, a simple holistic *top-down process, consisting of only a single run of the CI algorithm, is as effective as exhaustive runs of the path-specific top-down process (once for each call path) yet with an exponential speed-up*[6].

4.1 Consequences of clean-up

The reason that a path-specific top-down process provides the highest level of precision is that it considers only individual call paths in isolation from each other. It should be evident that any derivation tree formed from assignments along only a single call path is realizable. We generalize this observation in the following definition.

Definition 1 Let D be a derivation tree formed by applying the CI algorithm to a program. Then, we say that the derivation tree D *matches* a call path α when a call site c is in α if and only if D contains a param-assignment from c.

For instance, among the three derivation trees shown in Figure 2, the trees (a) and (b) have matching paths 1 and 2, respectively. Tree (c) does not match any single call path since the param-assignments are from 1 and 2, both to procedure bb(). From the definition, it is apparent that D matches a *unique* call path α. It is completely determined by the set of param-assignments in D. This also implies that any subtree of D matches a *unique* subpath of α.

Lemma 1 Let D be a derivation tree with a matching call path α. Then, it is interprocedurally realizable.

[4] Thus, it is required to protect the points-to information of a (and also c), which is already complete at the level of aa†(). Otherwise, surprisingly, context-sensitivity is *entirely* lost. This is why we have commented out the procedure calls in bb†() in the diagram on the previous page. From this perspective, the representation in the diagram must be understood as a topologically sorted modules with its own output-input transfer mechanism, rather than an actual program following the usual procedure-call semantics.

[5] Therefore, the output-to-input transfer is realized purely following the usual procedure-call semantics. (A modularized view of programs is no longer required.) Note that in Subsection 3.3, without the clean-up step, bb‡() still contains the side-effect production, therefore, an explicit separation from aa†() was necessary.

[6] A similar phenomenon (exponential speed-up) can be also found in distributive dataflow analysis [1, 25] and type-based polymorphic flow analysis [24, 17]. However, in these cases, the cause of the speed-up is inherent in the analysis problem itself. In ours, such an opportunity is manufactured by transforming programs into a *nice* form.

To be interprocedurally spurious, two dataflows formed by two different call paths leading to a single procedure must participate in D. However, since each subtree of D matches a subpath of α, such a case never arises. □

We will demonstrate that the CI algorithm applied to a side-effect-free program only produces a derivation tree with a matching call path. Since a program is already side-effect-free after the enhanced bottom-up process, this directly justifies our simple top-down approach, implemented as a single run of the CI algorithm.

Since we only consider acyclic ones, a call graph defines a natural partial order $\sqsubseteq$ between procedures. We also extend this partial order to work with variables defined in procedures. For instance, if pp is some descendent of qq and they define the variables u and v, respectively, then $pp \sqsubseteq qq$ implies $u \sqsubseteq v$. In such a case, we also say that there is a call path from v to u. In the following definition, $e(u)$ denotes some expression that contains a variable u.

Definition 2 A derivation tree of $e(u) := e'(v)$ formed by the CI algorithm is said to be a side-effect production if and only if an assignment from a strict descendant[7] of the *home* procedure of u (the procedure that declares `u`) appears in the derivation tree. Moreover, a program is said to be *side-effect-free* if the CI algorithm does not generate any side-effect productions.

Lemma 2 If a program is side-effect-free, then any derivation tree D of $e(u) := e'(v)$, produced by applying the CI algorithm to the program, always has a matching call path from v to u.

The proof is performed by applying structural induction on the derivation tree D, whose inductive definition is from Figure 1. In the following proof, A denotes the assignment set resulting from a context-insensitive aggregation of the program.

1. When the last small-step execution taken in D is rule (1), then, $e(u) := e'(v)$ is an element of A. If it is for parameter passing, D trivially matches a call path that consists of a single call site, from which the param-assignment originates. Otherwise, since u and v are from the same procedure, it matches an empty call path.

2. When the last small-step execution taken in D is rule (2) in the following form:

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

   Let D_1 and D_2 be the derivation trees of $u := w$ and $w := \&v$, respectively. Then, from the induction hypothesis, D_1 and D_2 have their matching call paths, namely, α from w to u and β from v to w, respectively. Then, D matches $\beta \cdot \alpha$, a call path from v to u.

3. When the last small-step execution taken in D is rule (4) in the following form:

$$\frac{w := \&u \quad *w := v}{u := v}$$

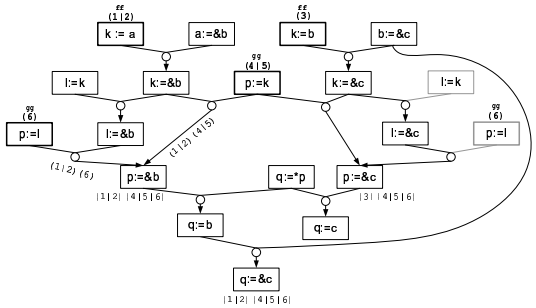
   The assignment $*w := v$ is never derived and w, v must be in the same procedure. Since $w := \&u$ is derivable, by the induction hypothesis, $w \sqsubseteq u$. Also, since the program is side-effect-free, from $*w := v$ and $u := v$, w must not be a strict descendent of u. Therefore, it enforces w and u to be in the same procedure and D matches an empty call path.

4. When the last small-step execution taken in D is rule (3) in the following form:

$$\frac{u := *w \quad w := \&v}{u := v}$$

   Let D_2 be the derivation tree of $w := \&v$. The assignment $u := *w$ is never derived. Thus, u,w must be in the same procedure, Therefore, D matches the same call path that D_2 matches. □

[7] If $pp \sqsubseteq qq$ and $pp \neq qq$, then we say that pp is a *strict descendent* of qq.




```
P3: ee()  {a:=&b; b:=&c; ff1(a); ff2(a); ff3(b);}
    ff(k) {gg4(k); gg5(k); gg6(l); l:=k; }
    gg(p) {q := *p;}
```

Figure 3: The derivation graph of the program `P3` formed by the CI algorithm and the result of its recovered context. The accumulated context information has been organized as regular expressions. The result `(1|2)(4|5|6)` at `p:=&b` denotes the six call paths along which it is derivable.

Therefore, given a side-effect-free program, the CI algorithm does not produce any spurious points-to information. As a consequence, after the enhanced bottom-up process, the simple top-down process (which consists of only a single, full program sweep of the CI algorithm) does not produce any spurious points-to information.

4.2 Context recovery

Among the two problems that we have raised against a path-unspecific top-down process, the spurious dataflow problem has been resolved by an additional clean-up step. The remaining issue is the context-loss problem.

We will sketch the outline of an *optional*[8] algorithm, named the *context-recovery* algorithm, which can efficiently con-

[8] This additional step is optional since the points-to information after the simple top-down process might be sufficient for many purposes.

$$\frac{u := \&v \wedge exposed(u)}{initial(u := \&v)} \quad (5)$$

$$\frac{u := v \wedge exposed(u)}{initial(u := v)} \quad (6)$$

$$\frac{*u := v \wedge hold(u)}{initial(*u := v)} \quad (7)$$

$$\frac{u := *v \wedge exposed(u)}{initial(u := *v)} \quad (8)$$

Figure 4: Initial assignments.

struct the missing context information[9]. The following is an oversimplified version of the context-recovery algorithm without efficiency consideration (for instance, grouping parm-assignments from the same call site, together). The input to the algorithm is G, the derivation graph from the CI algorithm performed within the simple top-down process.

```
1   For each parm-assign p in G
2       For each assign a reachable from p
3           Accumulate p's call site into a.
4   For each assign a in G
5       Postprocess call sites accumulated in a.
```

Figure 3 depicts a run of the context recovery algorithm for a sample program. After the first loop (lines 1-3) finishes, the call sites accumulated in each assignment $e(u) := e'(v)$ form a subgraph G' from v to u (from Lemma 2). Since each derivation tree embedded in the derivation graph is constructible, and also since each derivation tree is interprocedurally realizable, each path from v to u in the subgraph G' is valid context information, from which $e(u) := e'(v)$ is derived.

The postprocessing step at line 5 is application-specific. For instance, to visualize along which call paths u points to v, the raw information (which is simply a set of accumulated call sites) would be sufficient. For alias queries between expressions it is desirable to process the raw information into an appropriate form, such as regular DAGs [29, 30], which can be formed in almost linear time for an acyclic graph, so that the path-set intersection query can be answered efficiently[10].

5. SIDE-EFFECT EXTRACTION

This and the following section complete the description of the enhanced bottom-up process introduced in Section 4. This section focuses on the side-effect extraction process, or *compaction* [22], which is the central component in our bottom-up process. The success of the top-down process largely depends on the quality of this process. First, the boundary of the extracted summary must be sharp enough to make the side-effect production easily removable from callees. Second, the extraction process should not blow up and eventually overload the bottom-up process, otherwise the effectiveness of our top-down process becomes irrelevant.

5.1 Initial assignments

To perform side-effect extraction, it is essential to understand how side effects are actually produced. In our setting, side effects are initiated by assignments[11] of the form $*u := v$ such that u points to variables non-local to the procedure under consideration. Variable u is said to be *holding* and assignment $*u := v$ is said to be *initial* in the sense that all back substitution is initiated from the assignment. Consider the following example.

```
P4: ii()     { ... ... }
    hh(k,l) {m:=k; n:=l; *m:=n; *m:=o;}
```

In this example, `k` and `l` are parameters and regarded as holding addresses of non-local variables by default. Because of the assignment `m:=k`, the variable `m` is also considered to be holding. As a consequence, these assignments `*m:=n` and `*m:=o` are initial. For the assignment `*m:=o`, since `o` is never updated anywhere, its has an empty effect. On the other hand, by performing back substitution from `*m:=n` through `m:=k` and `n:=l`, the extraction process produces its net effect `*k:=l`. It is apparent that the external behavior of `hh`*`()`, the summary of `hh()`, is identical to that of the original procedure `hh()`.

`hh`*`(k,l) {*k:=l;}`

5.2 Exposed variables

The extraction process is complicated by the appearance of the address operator, particularly when the address of a local variable is assigned into a non-local variable. Consider the following example.

```
P5: jj()     {kk1(&a,&a); kk2(&b,&c);}
    kk(k,l) {*k:=&m; m:=l; **l:=k; n:=m;}
```

In this example, at the assignment `*k:=&m`, the address of the local variable `m` is potentially assigned into a nonlocal variable. We call such a variable *exposed*[12] in the sense that its address is exposed to nonlocal variables. (A variable not exposed is called *hidden*.) To replicate this side effect into callers, we *promote* local variable `m` so that the nonlocal variables can point to it at the levels of callers, in this case `jj()`, instead of `kk()`. (The details of such promotion will be explained in Subsection 6.2.)

Subsection 5.1 explained that an assignment $*u := v$ is the only possible source of side effect production. However, since we will promote exposed variables up to callers, a direct write into an exposed variable is also a side effect (in a non-standard way).

In the example `P5`, the only write into an exposed variable is the assignment `m:=l`. Since `**l:=k` is also a side effect production by itself, the extraction process can exclude only `n:=m` from the summary of `kk()`. The following is the resultant summary that will be promoted into `jj()`.

`kk`*`(k,l) {*k:=&m; m:=l; **l:=k;}`

[9] This algorithm, unfortunately, is not applicable to a derivation graph for an arbitrary run of the CI algorithm: It only works with a derivation graph without side effects. Therefore, it cannot be used to construct missing context information for neither a context-insensitive points-to analysis nor a modular points-to analysis without the clean-up step.

[10] The subscripts of variables, introduced by specialization during the bottom-up process, are also a part of context-information and must be considered together.

[11] In the presence of global variables, a plain assignment can also cause side effects. It can be easily and exactly handled by making global variables exposed by default.

[12] It is equivalent to the well-studied concept of *escape*, for instance, in [2, 5, 6, 9, 16, 31].

$$\frac{u := \&v \wedge exposed(u)}{exposed(v)} \quad (9)$$

$$\frac{*u := v \wedge v := \&w \wedge hold(u)}{exposed(w)} \quad (10)$$

$$\frac{param(u) \vee exposed(u)}{hold(u)} \quad (11)$$

$$\frac{u := v \wedge hold(v)}{hold(u)} \quad (12)$$

$$\frac{u := \&v \wedge exposed(v)}{hold(u)} \quad (13)$$

$$\frac{u := *v \wedge hold(v)}{hold(u)} \quad (14)$$

Figure 5: Holding and exposed variables.

5.3 Overall process

With all of the key concepts in the extraction process having been introduced, this section describes the overall process. The extraction algorithm begins by making the procedure of interest *closed* under the inference rules of Figure 1. Closure simplifies the determination of the holding and exposed properties easier by making all purely local dataflow explicit.

After the closure is formed, the extraction algorithm runs the inference rules in Figure 5 to determine any holding and exposed local variables. For the detailed explanation of these rules, we refer readers to [22]. Using this information, it finds all the initial assignments, using the inference rules in Figure 4, which are the sources of all the side effects. Finally, back substitution is performed to prune unnecessary assignments and to compacts side-effect productions[13].

The following is the most important invariant kept by the extraction process: *All writes into nonlocal variables, including exposed variables, will be reproduced by the resultant summary.* In certain cases, hidden variables may be included in the summary as intermediates for multiple dereferences such as `a:=*b, b:=*c`.

6. ENHANCED BOTTOM-UP PROCESS

The enhanced bottom-up process transforms arbitrary programs into a side-effect-free form without altering any points-to information. There are two steps. The first step is the compaction-based bottom-up process, which replicates side effects into callers. As long as the side effect extraction process results in an exact summary of a procedure, this step does not alter points-to information.

The second step is the clean-up process that removes all the initial assignments from each procedure. Since, by definition, all side-effect productions are initiated from initial assignments, it is clear that this step will make the procedures completely side-effect free. The only issue is whether this process alters the points-to information.

6.1 Clean-up step

After all side-effect productions have been replicated into callers, the clean-up step removes all the initial assignments thus completely converting a program into a side-effect-free form. Among the cases in Figure 4, the rules (5), (6), and (8) handle exposed variables and its implication will be explained in the following subsection.

All the remaining initial assignments are in the form of $*u := v$, as identified by the inference rule (7). In this case, since the callee is closed under the inference rules of Figure 1 (the side-effect extraction process needs it to determine holding and exposed properties of variables), all its local effects have been replicated by other assignments. Therefore, we can remove them from the callee without affecting the final points-to set derived for the callee.

6.2 Handling exposed variable

An exposed variable is treated as an extra parameter of its home procedure[14]. During the side-effect replication (inlining) step, at each call site, a specialized version of the exposed variable is introduced as a new local variable of the caller. After the clean-up step, all updates made to the specialized versions of the exposed variable become available to the original exposed variable in the callee through the usual parameter-passing mechanism. This is the only update into the original exposed variable after the clean-up step[15]. The following shows the results after applying this process to example `P5` in Subsection 5.2, where `m` was the only exposed variable.

```
P5': jj'() {
         k1:=&a; l1:=&a; *k1:=&m1; m1:=l1; **l1:=k1;
         k2:=&b; l2:=&c; *k2:=&m2; m2:=l2; **l2:=k2;
         kk'1(k1,l1,m1); kk'2(k2,l2,m2);
     }

     kk'(k,l,m) {
         n:=m; /* *k:=&m; m:=l; **l:=k; */
     }
```

Note that the signature of `kk'()` now contains `m` as an extra parameter. Also, all the initial assignments in `kk()`, including `m:=l`, have been removed. Instead, since the summary `kk*()` is inlined in `jj'()`, the side effect productions were replicated in `jj'()`. Therefore, the points-to information of the exposed variable `m` is complete in `jj'()` except that they are distributed into m_1 and m_2.

In `kk'()`, `n:=m` is still retained, even after the clean-up step, since its effect is purely local to `kk()` (therefore, not initial). In such a case, it is important that the points-to information of `m` is complete at the level of `kk'()`, too. In the example above, it is realized solely through the usual parameter-passing mechanism for the variable `m`. As a consequence, the removal of initial assignments, originated from the exposed variable `m`, does not affect any procedure.

[13] Exposed variables directly affect the size of summaries. From experience, few local variables, including address-taken ones, are found to be exposed. On the other hand, heap objects are more frequently exposed. In the experiments, exhaustive heap-object specialization was problematic for 4 benchmarks out of 19 tested. For those, specialization was limited by two generation (Figure 8).

[14] On the other hand, the inlined hidden variables are completely disconnected from their home procedures; they are purely place holders. In fact, [15] treats all the inlined variables, including exposed ones, in this way. This treatment requires all updates into exposed variables to remain at the level of their home procedures. In fact, it is a critical difference between [15] and ours.

[15] After the enhanced bottom-up process, the original exposed variable is never visible to callers, neither directly nor indirectly, and this update is not a side-effect production.

7. EXPERIMENTAL SETUP

While the focus of this paper is efficient, accurate computation of pointer values, it is important to overview aspects of the complete experimental system used to generate the empirical results.

7.1 Field Sensitivity

Many of the experiments make use of field information because such information can greatly refine the analysis results. Despite these benefits, since the C language is not type-safe, the analysis system must be somewhat distrustful of the field information or else it may obtain unsafe results for some programs. Our implementation of field sensitivity tries to handle as many abuses of the C language as possible in a safe and precise way.

For *field sensitivity* (FS), our analysis is offset-based, as opposed to using field names as in [23]. Each variable is composed of a size and accessed offsets. The size of named variables are limited to the size of the defined type. The size of heap objects are set to the size of the largest type in the program[16]. The accessed offsets for each variable are initially set to empty, but are adjusted throughout the analysis process as new updates are discovered. Thus the analysis process may detect accessed offsets beyond those explicit to the variable's defined type. Finally, the analysis system can disable field sensitivity for individual variables and will exercise this ability to further improve safety or speed (at the expense of precision).

Aside from improving safety, our mechanism for field-sensitivity permits variable-sized assignments for object accesses. For example, direct or indirect structure-to-structure copying is represented by a single large assignment instead of being discretized into multiple pointer-sized ones.

7.2 Context Sensitivity

Heap object specialization can improve the precision of a pointer analysis algorithm. To evaluate the benefit, two different settings of context sensitivity are used in the experiments, one with heap object specialization (HS) and the other without (HI). In the first, the heap objects are treated similar to local variables and become specialized to a creation call path. In the second, heap objects are treated as global variables and only a *fixed* number exist, one for each allocation routine[17].

Even in a context-sensitive algorithm, global variables are not subject to variable specialization and retain the same values across all possible call paths. Hence, in our framework, they are handled through a separate *context-insensitive* mechanism and excluded from the bottom-up process. This isolates the global variables from the compaction process, allowing the summaries to capture only the context-sensitive effects through specializable dataflow.

8. EMPIRICAL RESULTS

To demonstrate the usefulness of our approach, we evaluated the presented techniques on all the C benchmarks from the SPEC92, SPEC95, and SPEC2000 suites except redundant ones. This breadth of benchmarks shows a wide range of analysis characteristics. The analyzer, compiled with debug information and no optimization, was run on a 2.8 GHz Pentium 4 computer with 1GB memory.

8.1 Scalability

One major concern in terms of scalability is the problem size growth caused by summary replication. Figure 6 demonstrates the effectiveness of our bottom-up process by comparing the theoretical growth of a full inlining process (upper graph) against empirical results using our compaction algorithm (lower graph). In the figure, procedure `main()` is always at a depth of one while leaf procedures are at the far right of a particular line spanning depths from 11 to 39. The y-axis is a logarithmic scale; thus, the upper graph extends to 10 trillion while the lower graph extends only to 100K.

It is apparent that the growth in our bottom-up process is no longer explosive. An interesting aspect of the lower graph is that the size of summaries does not grow monotonically as it approaches `main()`, instead, it varies almost independently of depth, increasing and decreasing according to compaction opportunities.

Figure 7 shows three sets of data. The height of each bar is a logarithmic plot of the run times of the CS-FS pointer analysis. The fraction of time spent performing the top-down process is shown by the fraction of each bar colored red (dark). Finally, the solid line plots each benchmark's size on a logarithmic scale centered at 10k lines of code (LOC). It is apparent that the top-down process is a light-weight part of the analysis process, as it rarely surpasses 10% of the analysis time.

253.perlbmk stands out as having both the longest total run time (12 minutes) and the most time consuming top-down process. The problem is that it has a number of large, identical pointed-to-by sets which substantially burdens the entire analysis process. We believe that it can be resolved by employing simplification [28, 26], in addition to those already selectively applied [20] and [13], or an alternative set representation [4].

8.2 Precision

The results of a series of analysis runs are shown in Table 8. Columns 1 and 2 show the total CS-FS analysis times without and with HS. Overall, the times are quite small given our chosen design point and HS does not always dramatically increase the analysis times.

Column 3 shows the reduction in the points-to set size observed when going from CI-FI to CI-FS. Column 4 shows the *additional* points-to set size reduction observed when moving from CI-FS to CS-FS. FS provides a noticeable, if not significant, benefit for most benchmarks. With respect to FS, CS is more inconsistent but clearly helps.

Columns 5, 6, and 7 are for the evaluation of heap object specialization. HS tries to use calling contexts to distinguish heap objects. In some cases a large number of heap objects can result, so four benchmarks (*) were limited to 2 generations of heap objects. A single generation means one step upward from the original allocation routine.

Column 5 lists the number of heap objects created. Column 6 shows the percentage of those heap objects whose points-to set is less than half that obtained by the HI algorithm. For 132.ijpeg, 99% of the specialized heap objects

[16] For heap objects, since the C language is not type-safe, it may be a cause of unsoundness. It can be remedied by mapping all the offsets larger than the maximum type size into a single symbolic offset at the end.

[17] In most context-insensitive algorithms, a distinct variable is assigned for each use of an allocation routine. However, within our framework, it is naturally realized through the mechanism of context-sensitivity.

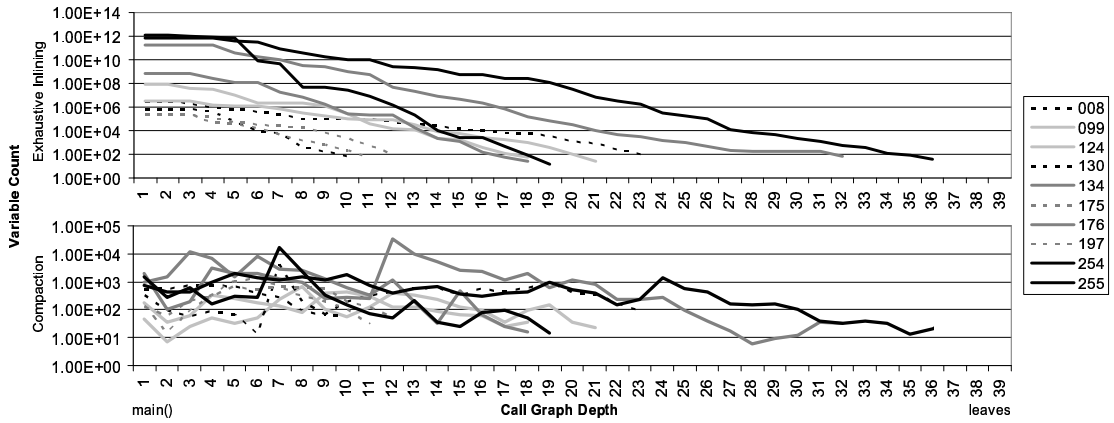


Figure 6: Log graph of module size versus call graph depth for exhaustive inlining and inlining using compaction.

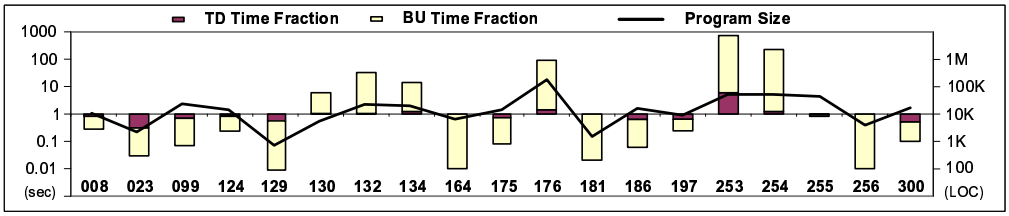


Figure 7: Log graph of total analysis time and benchmark size. Bar divisions show *fractional* (not logarithmic) split between top-down and bottom-up times.

had their points-to sets reduced by more than half. Column 7 shows a similar percentage, but instead for the number of objects that each heap object has pointing to it. For example, 124.m88ksim creates 10 heap objects, 70% of which have half as many objects pointing to them when compared to the HI result. (The higher the percentages in Columns 6 and 7, the more the heap specialization helped.)

Heap cloning was particularly beneficial to 132.ijpeg because it makes many indirect function calls through heap-allocated objects. Additionally, much of the heap allocation itself occurs through indirect function calls. Heap specialization permitted the construction of a far more accurate call graph. 254.gap shows a very limited amount of heap specialization. 254.gap uses `NewBag` to perform custom allocation and garbage collection through a global pool. We feel that, with some additional information, heap specialization can treat these custom allocation routines in a way similar to `malloc`.

9. RELATED WORK

Points-to analysis has been extensively studied in literature. We refer readers to [19] for extensive lists of previous work. In particular, context-sensitive points-to analyses have been studied in [23, 12, 32, 7, 8, 14, 11, 15]. Among them, [7, 8, 15] takes a modular approach. However, none of their top-down processes are both scalable and precise when the points-to information across all (or many) call paths is required. The algorithm in [15] is most similar to ours. It is based on the same inclusion-based intraprocedural setting. Procedures are summarized and compacted yet in the form of constraints (equivalent to our assignments). The major difference lies in the top-down process. They employ a path-unspecific top-down approach which causes precision degradation. Also, in many benchmarks, their top-down process takes longer than the bottom-up process.

For the programs with structural types and no implicit dataflow formed by alias, type-based polymorphic flow analyses in [24, 17] provide an elegant solution to the spurious interprocedural dataflow problem. An interesting aspect of these works is that even recursive procedures are handled polymorphically (by let-polymorphism). The context-sensitive points-to analysis in [14] has applied these approach to the C language, which does allow nonstructural types and implicit dataflow, by employing an equivalence-based intraprocedural setting [27].

10. CONCLUSION

We have proposed a fully context-sensitive yet very efficient holistic top-down process. When combined with our previously proposed, efficient and exact bottom-up process, it results in a scalable and precise modular points-to analysis. The key mechanism leverages the exactness of our bottom-up process to convert a program into a side effect free form. Empirical results support the scalability of our modular points-to analysis, in particular, that of the proposed top-down process. The benefit from context sensitivity and heap-object specialization has been demonstrated by an extensive study across a large set of benchmarks.

In the near future we plan to perform more detailed experiments, including a measurement of the benefit that the context-recovery algorithm provides for alias queries. Also, there is much interest in delineating exactly where and how context sensitivity benefits real programs. Once we are able to generalize the mechanism behind them, by exploiting the flexibility of the implemented framework, we plan to develop an adaptive balancing between precision and scalability.

	1	2	3	4	5	6	7
Benchmark	CS-FS HI (sec)	CS-FS HS (sec)	FS PtsTo Red. (%)	CS-HI PtsTo Red. (%)	Heap Object Spec.		
					Obj Count	<1/2 PtsTo (%)	<1/2 PtsBy (%)
008.espresso	1	*6	7	39	633	10	85
023.eqntott	1	1	13	-	9	66	44
099.go	1	1	-	-	0	-	-
124.m88ksim	1	1	80	3	10	-	70
129.compress	1	1	-	-	0	-	-
130.li	6	2	23	-	9	33	44
132.ijpeg	32	2	91	9	655	99	100
134.perl	14	*205	83	11	52	-	12
164.gzip	1	1	45	2	9	-	55
175.vpr	1	1	9	39	319	51	57
176.gcc	89	*160	88	2	52	71	75
181.mcf	1	1	85	-	7	83	86
186.crafty	1	1	8	33	13	-	93
197.parser	1	1	90	-	2	50	50
253.perlbmk	738	*912	32	3	64	-	2
254.gap	229	232	25	-	4	-	50
255.vortex	1	1	56	-	73	-	100
256.bzip2	1	1	13	-	11	-	91
300.twolf	1	1	-	-	183	-	94

* Heap specialization limited to 2 heap generations.

Figure 8: Analysis times, points-to set size reduction, and heap objects that benefit from specialization across SPEC92, SPEC95, and SPEC2000 integer benchmark suites

11. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1986.

[2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *SAS*, 1999.

[3] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D thesis, DIKU, Unversity of Copenhagen, 1994.

[4] Marc Berndl, Qndej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI*, 2003.

[5] Bruno Blanchet. Escape analysis for object oriented languages. Application to Java. In *OOPSLA*, 1999.

[6] Jess Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA*, 1999.

[7] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *POPL*, 1999.

[8] Ben-Chung Cheng and Wen-mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, 2000.

[9] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreenhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA*, 1999.

[10] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, 2000.

[11] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, 2001.

[12] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, 1994.

[13] Manuel Fähndrich, Jefrrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, 1998.

[14] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.

[15] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS*, 2001.

[16] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for Java programs. Technical report, Microsoft Research, 1998.

[17] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. *Lecture Notes in Computer Science Volume 2053*, 2001.

[18] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *PLDI*, 2001.

[19] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[20] John E. Hopcroft. An *n*log*n* algorithm for minimizing the states of a finite automaton. *The Theory of Machines and Computations*, pages 189–196, 1971.

[21] Marc Shapiro II and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL*, 1997.

[22] Hong-Seok Kim, Erik. M. Nystrom, Ronald D. Barnes, and Wen-mei W. Hwu. Compaction algorithm for precise modular context-sensitive points-to analysis. Submitted to CC2004. Also available on the authors' website as a techical report.

[23] W.A. Landi and Barbara G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *PLDI*, 1992.

[24] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, 2001.

[25] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[26] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, 2000.

[27] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *POPL*, 1996.

[28] Zhendong Su, Manuel Fahndrich, and Alexander Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *POPL*, 2000.

[29] Robert E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28:594–614, 1981.

[30] Mark Wegman. Summarizing graphs by regular expressions. In *POPL*, 1993.

[31] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, 1999.

[32] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.