Compaction Algorithm for Precise Modular Context-Sensitive Points-to Analysis

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

IMPACT Technical Report, IMPACT-03-03
Center for Reliable and High Performance Computing
University of Illinois, Urbana-Champaign

{nystrom, hskim, rdbarnes, hwu}@crhc.uiuc.edu

ABSTRACT

Points-to analysis is a critical component of optimization and software engineering tools. A higher degree of precision in this analysis can significantly benefit such tools if it can be provided with a reasonably bounded analysis cost. The focus of this work is to enable a fully context-sensitive modular points-to analysis under the flow-insensitive and inclusion-based intraprocedural setting. Such a modular approach reduces the interprocedural problem to a series of intraprocedural ones. The main source of difficulty with such an approach is that the size of the module grows very quickly as they are built up as the bottom-up phase of the analysis proceeds.

The main contribution of this work is compaction, an algorithm that produces a concise-yet-equivalent summary of a procedure's module. It allows the modular points-to analysis to enjoy the full power of context sensitivity in a scalable fashion. Through empirical results, both the necessity and the effectiveness of the proposed techniques are demonstrated.

1. INTRODUCTION

Context sensitivity is becoming increasingly more important because modern programming practices exasperate the problem of spurious data-flow between independent calling contexts by encouraging code reuse. In these situations, a context-sensitive program analysis achieves a higher degree of precision than its context-insensitive counterpart by disallowing unrealizable interprocedural data-flow.

In the domain of points-to analysis, the significance of context sensitivity is increased further when programs make use of heap allocation. For example, consider a program that makes all heap allocations via a program-specific wrapper around the standard library allocation routines. Without context sensitivity the entire heap is abstracted into a single static entity through which an enormous amount of spurious data-flow can occur. A carefully designed context-sensitive points-to analysis can resolve such a problem by specializing heap objects according to call paths.

Some empirical results have not supported the significance of context sensitivity in the precision of pointer analysis [26, 14, 11]. However, the first two algorithms [26, 14] are not tested with large programs, since they do not scale well, while the last algorithm [11] achieves scalability by sacrificing a nontrivial amount of precision. It is unclear how much of the result is affected by the precision degradation. Our experiences within our own compilation framework have shown that the benefit can be large but varies substantially from program to program, being nominal for some while achieving a large reduction (30-80%) in points-to size for others. The benefit is typically increased for large programs.

One methodology for achieving context sensitivity is through *modularization* [7, 8]. A modularized analysis consists, at least conceptually, of two phases. Guided by the call graph, the first phase transforms each procedure into a single, self-contained *module* that encapsulates all of the internal points-to relationships of the procedure along with all of the side-effects of its immediate and transitive callees. This reduces the *inter*procedural problem to a series of *intra*procedural problems. The second phase obtains the final points-to results by analyzing the modular version of the procedure along with all potential inputs to that procedure. The complete process is covered in detail in Section 2.2.

Exhaustive inlining of callees would be a sufficient way to account for the side-effects of a procedure's callees but, clearly, this would be far from concise and quickly explodes in size. The key issue in a modular approach is finding an effective way to represent the side-effects of each callee in the form of a concise *summary*.

Depending on the desired analysis goal, a summary can be formed either in a conservative way or in an exact way. Orthogonally, a summary may be valid across all possible calling scenarios or partial thus may need to be continuously updated to cover unexplored situations. Conservative or partial methods may, in some cases, allow for the formation of smaller summaries. However, it may also suffer from reduced precision, increased overhead, or necessitate multiple summarizations of the same procedure. Contrasting with such approaches, *we present an technique called* ***compaction*** *that obtains* ***exact*** *and* ***universally applicable****, yet, in practice, small procedure summaries at low costs.*

2. PRELIMINARIES

This section gives a brief and intuitive description of our entire modular points-to analysis methodology. This background will provide a foundation on which more detailed discussion central to compaction can be constructed.

2.1 Intraprocedural setting

Intraprocedurally, we have made our modular points-to analysis operate in a flow-insensitive manner for the following reasons: it greatly simplifies our initial investigations, and it is less likely to affect precision significantly, as reported by [18, 19]. Among many different forms of flow-insensitive approaches [4, 10, 17, 21, 27], we have chosen the inclusion-based approach [4], which is the most precise among those, with the goal of keeping the cost within a reasonable bound.

Offset sensitivity, alternatively, has an important effect on the resultant precision and is readily compatible with our compaction approach. However, since the focus of this work is on context sensitivity, we consider offset expressions only as part of future work in Section 9.

Given these simplifications, the following four kinds of assignments are sufficient for our discussion:

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

where u, v are variables, & the address operator, $*$ the dereference operator. The intuitive meaning of those operators are the same as it is in the C language.

Given a set of assignments A, the points-to deduction can be described in the style of [17] as follows:

$$\frac{\alpha \in A}{A \vdash \alpha} \tag{1}$$

$$\frac{A \vdash u := v \quad A \vdash v := \&w}{A \vdash u := \&w} \tag{2}$$

$$\frac{A \vdash *u := v \quad A \vdash u := \&w}{A \vdash w := v} \tag{3}$$

$$\frac{A \vdash u := *v \quad A \vdash v := \&w}{A \vdash u := w} \tag{4}$$

If $A \vdash \alpha$, then we say that α is derivable in A. If α is an address assignment in a form $u := \&v$, we say that u points to v in A.

2.2 Interprocedural setting

As briefly mentioned in Section 1, we take a modular approach to transforming an inherently interprocedural problem into many modularized intraprocedural subproblems. Such a technique forms intraprocedural entities, called modules, that encapsulate all interprocedural actions taken inside a procedure, including all the side effects from its callees in a transitive fashion. Figure 1 presents a graphical overview of the entire process and demonstrates where the compaction algorithm fits within it. Figure 1(a) depicts the procedure call graph for an example program to be analyzed.

The most intuitive way to construct a module is through procedure inlining, as shown in Figure 1(b). In the figure, inlining proceeds upward from the bottom of the call graph until it reaches the root procedure, which, in this case, is procedure `A`. The following are the highlights of this process:

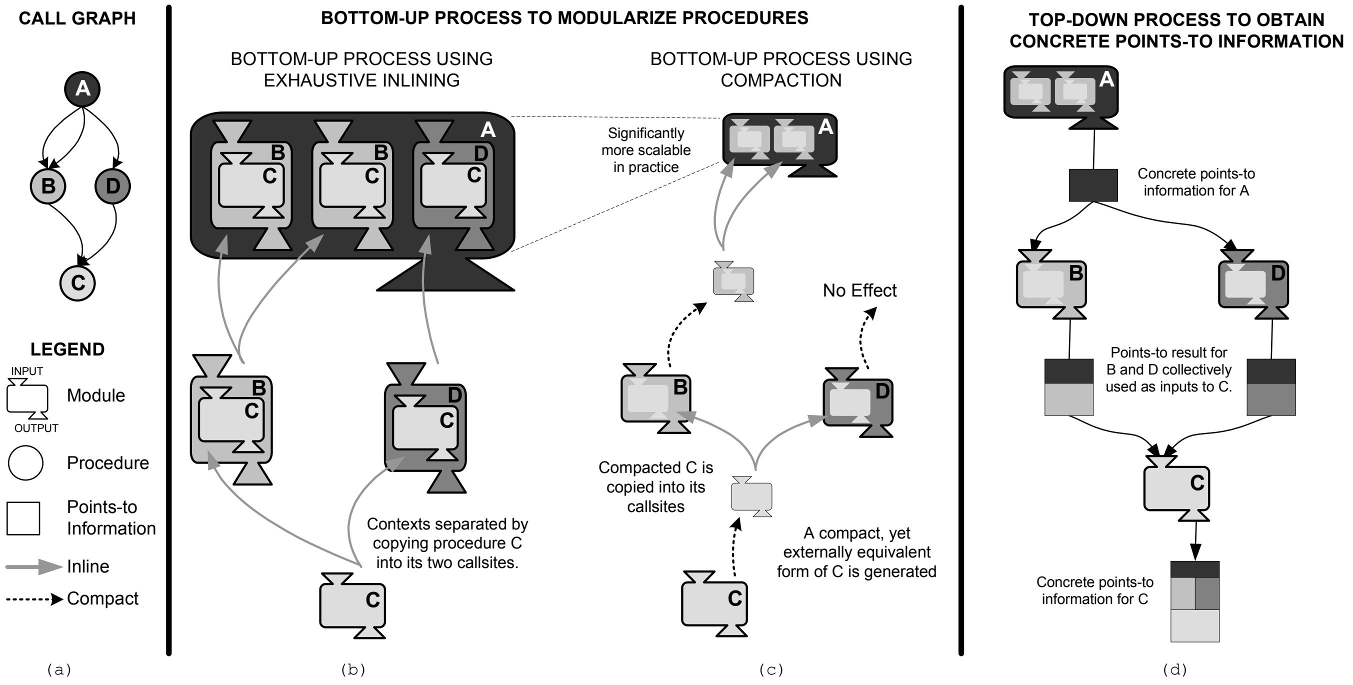
CALL GRAPH
BOTTOM-UP PROCESS TO MODULARIZE PROCEDURES
BOTTOM-UP PROCESS USING EXHAUSTIVE INLINING
BOTTOM-UP PROCESS USING COMPACTION
TOP-DOWN PROCESS TO OBTAIN CONCRETE POINTS-TO INFORMATION
LEGEND
INPUT
Module
OUTPUT
Procedure
Points-to Information
Inline
Compact
Significantly more scalable in practice
No Effect
Contexts separated by copying procedure C into its two callsites.
Compacted C is copied into its callsites
A compact, yet externally equivalent form of C is generated
Concrete points-to information for A
Points-to result for B and D collectively used as inputs to C.
Concrete points-to information for C
(a)
(b)
(c)
(d)

Figure 1: Modular process overview (a) Call graph (b) Modularization through exhaustive inlining (c) Modularization using compaction between inlining steps (d) Top-down resolution of concrete points-to information.

- Modularization begins from the leaf of the call graph, in this case, procedure C. Since it does not have any callees, it is already in a modular form.
- Procedures B and D call procedure C. To modularize them, C is inlined into each. For context sensitivity, two distinct instances of C are inlined into B and D, respectively.
- Procedure A calls B twice and D once. To modularize A, the modules of B and D are inlined. As consequence, the module of A contains three versions of C, which corresponds to the number of paths from A to C.

The modularization process shown in Figure 1(b) is called *exhaustive inlining*. It is apparent that a high level of context sensitivity is achieved by such a process. However, it is also apparent that the module size grows exponentially. Therefore, to be practical, modularization requires a technique like compaction to reduce the size of the modules. Intuitively, compaction is possible for two reasons. First, many actions that occur in a callee do not affect its callers. Second, even for those actions that do affect callers, the effects likely can be more concisely represented. The process of compaction is depicted in Figure 1(c). Since compaction is performed before a callee summary is inlined into a caller, module growth is reduced. In practice, our technique proves to be effective at controlling module growth.

Once all the modules are constructed, the top-down phase, depicted in Figure 1(d), generates the actual point-to information.

- Since procedure A has no callers, there is no calling context—its points-to information is computed solely from its module.
- Since B is called within A, it needs some information from A to compute its own points-to information. The points-to information from A is fed into the module of B to complete the setup for computing the points-to information of B. D is handled in an identical fashion.
- Since C is called from both B and D, both are needed for C's calling context. The points-to information from B and D are fed into the C's module, and then its points-to information then can be computed.

The example shown in the process in Figure 1 assumes a known, acyclic call graph. However, in real programs, a call graph is not known a priori (due to indirect calls) and may contain cycles (due to recursion). Within our framework, such complications are resolved in a manner similar to other works such as [7, 8].

Indirect calls are handled in an optimistic iterative manner. The algorithm begins with a call graph resulting only from direct calls. At the end of each iteration, the new points-to information updates the call graph with new indirect call targets. Iterations continue until no more updates on the call graph are made. Recursion is handled by merging recursive procedures into a single procedure. All recursive calls are simulated context-insensitively; only assignments for parameter passing are added.

The rest of the paper assumes a fixed acyclic call graph and solely focuses on the compaction process.

2.3 Module representation

In this work, all variables are classified as either global variables or local variables. At this point, we will not explicitly consider heap objects; such objects can be treated either as global variables or as special local variables, whose lifetime is not bounded by their home procedures. The first option yields a single heap object per allocation site. The second introduces specialized heap objects per allocation site, each corresponds to a call path to the allocation site. In Section 7, we will show some preliminary experimental results comparing these two options.

Since modularization completely encapsulates all interprocedural actions for a procedure, a module can be seen simply as a procedure without any callees. The following is a syntactic representation of a module:

$$M\ (\vec{P})\ \textbf{with}\ L\ \textbf{in}\ A \tag{5}$$

where M is the module being defined, $\vec{P}$ a vector of parameters, L a set of local variables that also contains all elements of $\vec{P}$, and A a set of assignments. Return values have been excluded from the module definition, since they can be simulated through the addition of an extra argument. (In the actual implementation, we have extended the algorithm described in this paper to work directly with return values.) The assignments in a module can only contain declared local variables, and global variables.

Consider the dynamic semantics of the C language. Some input memory state exists before the execution of a procedure begins. Once the procedure has completed its execution, some final output memory state has been created. Modules can be regarded in a similar way. In this situation, the inputs and outputs are abstractly represented as sets of address assignments, which we call *points-to graphs*. In the case of an input, the points-to graph is called a *calling context*.

Given a calling context and the module M, defined as in (5), the output computation is performed as follows:

$$M, C \models \alpha \text{ if and only if } C \cup A \vdash \alpha \tag{6}$$

If $M, C \models \alpha$, we say that α is derivable within module M given calling context C. Also, if α is an address assignment in the form $u := \&v$, we say that u points to v within module M given calling context C.

3. CHALLENGES

To motivate and focus the development of the compaction algorithm described in Section 5, the difficulties in performing compaction are articulated in this section. As observed in Section 2, not all the actions taken in a module directly affect callers. Consider the following example:

```
E(p,q) with {u} in {*q:=u, u:=p}
```

Given a calling context `{p:=&a, q:=&b}`, the following set of assignments are newly derived within the module: `{u:=&a, b:=u, b:=&a}`. Among these, the only assignment meaningful from the perspective of the caller is `b:=&a`. In this paper, we call such an assignment a *side effect*. All other assignments are called *intermediate* assignments.

From this perspective, two modules are said to be *equivalent* if they produce the same set of side effects given the same calling contexts. Also, in such a case, one module is said to be a *summary* of the other module. For instance, it is apparent that the following is is a summary of module `E`:

```
F(p,q) with {} in {*q:=p}
```

3.1 Back substitution

The main goal of this paper is to present an efficient algorithm to construct a smaller yet equivalent summary of a module. One strategy toward this goal is to minimize the number of local variables within a summary by performing back substitution. For instance, the summary `F` is obtained from `E` by performing back substitution on assignment `*q:=u` from its source `u` towards assignment `u:=p`. This back substitution produces assignment `*q:=p`. Since no further back substitution can be performed, the effect of `*q:=u` within module `E` is completely subsumed by `*q:=p`. Therefore, in a summary of module `E`, it is the only assignment required.

In certain cases, back substitution can also be made involving dereference assignments as follows:

```
G(p,q) with {u} in {u:=p, *u:=q}
H(p,q) with {} in {*p:=q}
```

For module `G`, back substitution can begin from the destination of `*u:=q` towards `u:=p` and result in `*p:=q`. A similar situation occurs when the dereference occurs in the right-hand side of an assignment.

In the examples so far, it has been possible to eliminate all of the local variables within the original module. However, in certain cases, back substitution may be unable to bypass all local variables. Consider the following example:

```
I(p,q) with {u,v} in {*p:=u, u:=*v, v:=*q}
```

In this case, none of the uses of `u` can be removed since our assignment syntax does not contain double dereference assignments like `*p:=*v` or `u:=**q`.

3.2 Implicit data-flow through aliases

The primary complicating factor we consider in this paper is that of implicit data-flow [1] induced by aliases. Within our setting, aliasing results solely from address assignments. For instance, $u := \&v$ makes $*u$ and v alias. Aliasing makes it difficult to remove local variables via back substitution because back substitution is only capable of handling *explicit* [2] assignments. Consider the following example:

```
J(p,q) with {u,v} in {u:=&v, *u:=q, *p:=v}
```

In this example, there is an implicit data-flow `v:=q` induced

[1]Informally speaking, a data-flow corresponds to a collective action that may span more than one assignment. For instance, from $u := v$ and $v := w$, data-flow $u := w$ is established.

[2]An assignment is *explict* if it is in a module or in a calling context. An assignment is *implicit* if it is not explicit and yet derivable. An implicit assignment becomes explicit by adding it into a module.

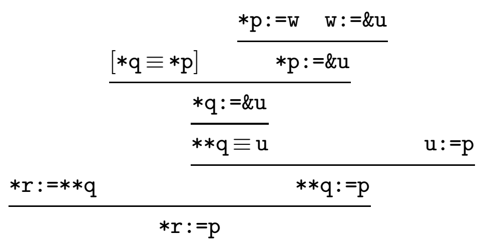

Figure 2: Informal visualization of the establishment of data-flow `*r:=p` within module `H` when `*p` and `*q` alias. The symbol ≡ is used to represent alias relationships between expressions.

by aliases between `*u` and `v`. Taking this into account, data-flow `*p:=q` can be established. Fortunately in this case, the data-flow `*p:=q` occurs regardless of calling contexts and is the only assignment needed in the summary. The following is equivalent to module `J`:

```
K(p,q) with {} in {*p:=q}
```

Note that the local variable `v` has been successfully removed, even though its address is taken. This elimination is possible because the aliasing of `v` is independent of calling contexts. However, in other cases, the nature of the aliasing may be impossible to determine without exact knowledge of the calling context. Consider the following example, where a multiple dereference is allowed in favor of clarity:

```
L(p,q,r) with {u,w} in
        {u:=p, *p:=w, w:=&u, *r:=**q}
```

Since the address of `u` flows into `*p`, aliases of `u` cannot be determined without consulting calling contexts. Consider a calling context where `*p` and `*q` alias, and `**q` and `u` alias. In this case, as depicted by Figure 2, a data-flow `*r:=p` is established. However, if `*p` and `*q` do not alias, such data-flow cannot be established.

The ability to derive the assignment `*r:=p` varies by calling context. This means that any data-flow involving `*r:=p` will be *context-dependent* and all the others *context-independent*. The challenge here is to encode this data-flow within a summary while retaining its context-dependence. In fact, as it will be clearer in Section 5, it is impossible to encode such dependence without allowing the address of local variable `u`, the main cause of such dependence, in the summary.

3.3 Necessity of back substitution

The second source of complication is including only the relevant results from back substitution, as well as finding the best locations from which to begin back substitution. One solution is to perform back substitution from every left dereference assignment, such as $*u := v$. However, this is inefficient, may miss many opportunities to produce smaller summaries, and may require a more complex algorithm that includes backtracking. Consider the following example.

```
M() with {u,v,w} in {*u:=v, v:=*w}
```

If back substitution begins blindly with `*u:=v`, it will next see `v:=*w`. Back substitution cannot continue through `v` because of the dereference on `w`. This means that the algorithm must decide whether or not to add the assignment. It would be a bad choice to add the assignment because the it is unnecessary. However, otherwise the algorithm would require backtracking, since it may turn out to be a necessary one later.

A similar situation can happen even when back substitution starts at a relevant location. Consider the following:

```
N(p,q) with {u,w} in {*p:=u, u:=*q, u:=*w}
```

If back substitution knows to begin from `*p:=u`, it will next see `u:=*q` which is a necessary assignment. It will proceed to `u:=*w`, resulting in the same dilemma as before.

The proposed algorithm resolves these two problems by computing necessity information before beginning back substitution.

4. RELATED WORK

A large body of research surrounds pointer analysis and it is important to delineate how the approach advocated by this paper fits within the continuum of other mechanisms.

4.1 Summary encoding

From the perspective of semantics, a procedure summary can be understood as a partial function betweens points-to graphs. Within our terminology, an input points-to graph corresponds to as calling context and an output points-to graph a set of side effects.

In our work, we utilize the computational aspects of partial functions to encode a summary: Taking points-to deductions as a semantic basis, our summary essentially describes how the output can be computed from the input. On the other hand, other context-sensitive points-to analyses in literature, such as [30, 7], represent a procedure summary as a mapping between abstract forms of points-to graphs. For instance, in [7], the input points-to graph is abstracted into alias pairs, whereas the output points-to graphs are abstracted into symbolic points-to graphs. Unfortunately, those algorithms are flow-sensitive and a direct comparison between their approach and ours is not possible. However, we can make a generic comparison between our computational approach and the mapping approach.

A mapping approach is usually inexact in the sense that the finite presentations in a mapping form can describe only a small set of partial functions without becoming conservative. Additionally, encoding and decoding processes are required to move between the concrete and abstract points-to graphs. Scalability is maintained by controlling the size of mapping by the degree of abstraction for input and output points-to graphs. Finally, the formation of the summaries is eager in the sense the algorithm must explicitly consider all the possible cases, which may affect how the output is produced.

On the other hand, our computational approach is exact because the summary represents the exact partial function without any loss in precision. Scalability is maintained by performing compaction, replacing a larger summary with a smaller yet equivalent one. Finally, our computational approach is lazy in the sense that the actual computation of the output is done only after a summary is inlined into a caller, at which point the actual input is available.

4.2 Polymorphic flow analysis

In the functional language domain, there has been significant interest in polymorphic flow analysis. Recently work such as [24, 16] have reduced the complexity to $O(n^3)$, where n is a size of a type-annotated program potentially exponentially larger than its type-erased counterpart. One key concept is shared by both algorithms—polymorphism is achieved through constraint instantiation rather than inlining.

There are two factors which make such an approach applicable. First, they exploit type information to reason and model information flow in programs. This implies that the calling contexts of a function have a fixed, known shape that is solely determined by type information. Second, there is no aliasing. This implies that the shape of intraprocedural data-flow is also fixed and independent to calling contexts.

The situation in context-sensitive points-to analysis is quite different. First, the shapes of calling contexts are not known a priori because this knowledge requires the points-to information itself. Second, the shape of the intraprocedural data-flow depends on calling contexts through the effect of aliasing. In general, the data-flow created by context-sensitive points-to analysis has a less regular structure and thus is more difficult to exploit.

One interesting difference between our work and [24, 16] is how recursion and indirect calls are handled. We handle recursive procedures context-insensitively by merging them into a single procedure. However, [24, 16] handle recursive procedures in a context-sensitive way. We also handle each instance of indirectly invoked procedures context-sensitively whereas [24, 16] allows polymorphism per each instance of a procedure appearance.

4.3 Constraint simplification

Simplification has been used in context-insensitive frameworks to improve efficiency [12, 25, 28]. There has also been significant interest in constraint simplification in the context of subtyping, for instance, [1, 23, 22]. In particular, the overall structure of the proposed algorithm is close to [22] in three ways. First, it forms a closure to make information more explicit. Second, it determines which information is relevant and throws away any irrelevant part. Third, it uses the DFA minimization algorithm [20] to merge equivalence information.

5. COMPACTION ALGORITHM

The compaction algorithm is roughly divided into three components: the first reduces the number of address-taken variables, the second determines where back substitution begins, and the third performs the actual back substitution.

The overall strategy of the compaction algorithm is to remove as many local variables as possible by performing back substitution. However, as described in Section 3, back substitution is largely ineffective in the presence of aliasing.

$$\frac{\mathit{global}(u)}{\mathit{opaque}(u)} \tag{7}$$

$$\frac{u := \&v \wedge \mathit{opaque}(u)}{\mathit{opaque}(v)} \tag{8}$$

$$\frac{*\,u := v \wedge v := \&w \wedge \mathit{hold}(u)}{\mathit{opaque}(w)} \tag{9}$$

$$\frac{\mathit{param}(u) \vee \mathit{opaque}(u)}{\mathit{hold}(u)} \tag{10}$$

$$\frac{u := v \wedge \mathit{hold}(v)}{\mathit{hold}(u)} \tag{11}$$

$$\frac{u := \&v \wedge \mathit{opaque}(v)}{\mathit{hold}(u)} \tag{12}$$

$$\frac{u := *v \wedge \mathit{hold}(v)}{\mathit{hold}(u)} \tag{13}$$

Figure 3: Transparency Detection.

In order to remove the bulk of the obstructions created by aliasing, our algorithm transforms as many address-taken local variables as possible into an address-free form [3]. This transformation is possible because the callers of a procedure are only affected by the subset of data-flow involving local variables that result in side effects. Therefore, address-free forms of local variables can be used as long as all implicit data-flow encoded by them is explicitly reflected in the summary.

In many, if not most, cases implicit data-flow can be exactly reflected through the addition of assignments, and an address-free form can be obtained through the deletion of certain address assignments. In the trivial case where no local variable addresses are taken, all of the assignments are explicit and context-independent. As shown in Section 3.2, taking a local variable's address may result in implicit, but still context independent data-flow. In this case, the implicit data-flow can be made explicit and the address assignment removed.

However, the removal of all address assignments is not always possible. As shown in Section 3.2, some assignments encode context-dependent data-flow. In these cases, the address assignment must either be retained (an exact solution) or all potential, implicit assignments must be made explicit regardless of context (a conservative solution). Our mechanism retains the address assignments in order to maintain precision.

[3]A variable `u` is said to be *address-taken* if the `&u` is used in a module or calling context. Otherwise it is said to be *address-free*. A local variable `u` is changed from address-taken to address-free when all assignments involving `&u` on the right-hand side are removed.

In practice, the detection of variables whose data-flow are context-independent is inexpensive. Additionally, the vast majority of local variables in real programs, regardless of whether their address is taken, are context-independent.

5.1 Transparency detection

The goal of this subsection is to show how to determine the local variables whose data-flow is context-independent. For convenience, local variables involved only in context-independent data-flow are *transparent* while all other variables are *opaque*. These concepts are similar to escape of objects, heavily discussed for object-oriented languages [2, 5, 6, 9, 15, 29].

The key to detecting transparent variables is recognizing the following: *Any local variables only pointed to by other local variables will have only context-independent data-flow, and thus be transparent.*

- If a local variable `u` is pointed-to only by other non-opaque local variables, all derived data-flow involving `u` can only involve local variables.
- If `u`'s data-flow involves only other local variables, its data-flow is context-independent.
- This implies every derivation involving `u` can be found through purely local points-to deduction.

Before attempting to detect transparent variables, intraprocedural points-to deduction is performed on the module using an empty calling context, and all derivable assignments are added to the module. This does not affect the equivalence of the module, and, by definition, all context independent data-flow must now be explicit.

Initially, all local variables are considered to be transparent and all global variables are considered to be opaque. The algorithm uses the inference rules in Figure 3 to determine which local variables are opaque; this process continues until the solution converges.

At this point, it is helpful to revisit how a local variable becomes opaque. Consider the following assignments: {`*u:=v`, `v:=&w`}. The net result of these two assignments is that the address of `w` will get assigned into anything to which `u` points. If it is possible for `u` to point to a non-local variable, then `w` is opaque. Additionally, if `u` points to an opaque local variable we conservatively make `w` opaque as well.

Since we do not have an actual calling context, the incoming pointer relations are still unknown. Some other mechanism must be used to determine whether or not a local variable points to a non-local one. The flow of external pointer values from an imaginary calling context is tracked using a property we call *holding*. Intuitively, a variable is said to be *holding* if it is theoretically possible for that variable to point to a non-local variable in some calling context. From this property it can be determined which local variables may be pointed to by an external variable, and thus are opaque. Any opaque variables are considered to hold, since their data-flow is unknown. By default, parameters and globals are considered to hold as well.

$$\frac{u := \&v \wedge opaque(u)}{initial(u := \&v)} \tag{14}$$

$$\frac{u := v \wedge opaque(u) \wedge hold(v)}{initial(u := v)} \tag{15}$$

$$\frac{*\,u := v \wedge hold(u) \wedge hold(v)}{initial(*u := v)} \tag{16}$$

$$\frac{u := *v \wedge opaque(u)}{initial(u := *v)} \tag{17}$$

Figure 4: Initial assignment detection.

The following example will be used to step through the inference rules of the proposed transparency detection algorithm in Figure 3:

```
P(p,q,r) with {u,v,w} in {u:=p, *u:=w, w:=&v}
```

In this example, the initial points-to deduction within the module does not produce any new assignments. The opaqueness of variable `v` can be thus be derived as follows. Using rule (10), variable `p` is a parameter and thus `holds`. Since `p` potentially holds an external address, `u:=p` means that `u` holds by rule (11). By rule (9), `v` is found to be opaque.

$$\dfrac{\dfrac{\texttt{u:=p} \qquad \dfrac{param(\texttt{p}) \vee opaque(\texttt{p})}{hold(\texttt{p})}}{hold(\texttt{u})} \qquad \texttt{*u:=w} \quad \texttt{w:=\&v}}{opaque(\texttt{v})}$$

5.2 Initial assignments

Once the transparency and opaqueness of all local variables is known, the address assignments involving transparent variables are pruned, leaving those variables address-free. Next, the compaction algorithm must chose the assignments from which to begin back substitution, as was illustrated in Section 3.

An assignment α is said to be an *initial* assignment if and only if there exists a calling context such that α is a *direct cause* of a side effect. Given a module M and calling context C, there are four kinds of direct causes:

1. If the address assignment $u := \&v$ was originally in M or C, then $u := \&v$ is a direct cause of $u := \&v$.
2. If the following derivation is possible, then $u := v$ is a direct cause of $u := \&w$.
$$\frac{u \overset{\star}{:=} v \quad v := \&w}{u := \&w}$$
3. If the following derivation is possible, then $*u := v$ is a direct cause of $w := \&x$.
$$\dfrac{\dfrac{*\,u \overset{\star}{:=} v \quad u := \&w}{w := v} \quad v := \&x}{w := \&x}$$

$$\frac{opaque(u) \ \vee \ param(u)}{input(u)} \tag{18}$$

$$\frac{initial(e := e')}{visit(e := e')} \tag{19}$$

Figure 5: Summary construction: initiation.

4. If the following derivation is possible, then $u := *v$ is a direct cause of $w := \&x$.

$$\frac{\dfrac{u \overset{\star}{:=} *v \quad v := \&w}{u := w} \quad w := \&x}{u := \&x}$$

Note that we have put symbol $\star$ on the direct cause for each case. Figure 4 presents our initial assignment detection algorithm, whose inference rules are parallel to the four cases described above.

5.3 Summary construction

The actual back substitution occurs as the summary construction process. The process starts with an empty summary. Gradually, as back substitution proceeds, assignments will be added into the summary until no more are needed. This summary construction algorithm exploits the following three concepts: *visit*, *deferral*, and *addition*.

1. An assignment is *visited* when the summary construction algorithm has chosen to simulate the consequences of that assignment.
2. A visited assignment is *deferred* by visiting other assignments in the hope of using those new assignments to simulate its consequences, thereby excluding it from the compact summary.
3. If a visited assignment must be included in the summary because its consequences will not take place without it, the assignment is *added* into the compact summary.

Figure 5 presents the initialization part of the summary construction algorithm. At this stage all the initial assignments are put into a work list so that they can be visited iteratively. The predicate *input* is used to indicate that the contents of a variable are unpredictable and cannot be back substituted.

Figures 6 and 7 describe the deferral process in detail. Each inference rule can be interpreted as follows: The upper part indicates the assignment that is about to be deferred along with the conditions under which it may influence the resultant summary. Should these conditions be met, the lower part indicates which assignment to visit and/or which assignment to add to the summary.

At the completion of this process, the compact summary will contain only those assignments added during the deferral process. This summary can be universally used in place

$$\frac{visit(u := v) \wedge v := \&w}{visit(u := \&w)} \tag{20}$$

$$\frac{visit(u := v) \wedge v := *w \wedge hold(w)}{visit(u := *w)} \tag{21}$$

$$\frac{visit(u := v) \wedge v := w \wedge hold(w)}{visit(u := w)} \tag{22}$$

$$\frac{visit(u := v) \wedge input(v)}{add(u := v)} \tag{23}$$

$$\frac{visit(u := \&v)}{add(u := \&v)} \tag{24}$$

Figure 6: Summary construction: deferral of plain and address assignments.

of the module that was summarized. Section 3 noted that care is needed to avoid adding unnecessary assignments to the summary during the back substitution process. The inference rules take into account the `hold` property (Initially derived for transparency detection) to avoid adding certain unnecessary assignments. Consider the example first introduced in Section 3:

```
N(p,q) with {u,v,w} in {*p:=u, v:=*q, u:=*w}
```

`*p:=u` is an initial assignment. When back substitution is performed, it can not go through `u:=*w`. The only rule seemingly applicable is (32), but it requires that variable `w` is `holding`, which it is not. Therefore, no rule is applicable and `*p:=u` is excluded from the summary.

6. MISCELLANEOUS COMPACTION

The compaction algorithm described in the previous section is generally effective but leaves misses a couple of important opportunities for further summary size reduction.

6.1 Redundancy elimination

An aspect in which back substitution cannot reduce the size of summaries is the detection and removal of redundancy obfuscated by intermediate variables. For example, the following procedure contains two redundant pairs of assignments.

```
Q(p,q) with {u,v} in {u:=*p, v:=*p, *q:=u, *q:=v}
```

Ideally, the generated summary should include only one pair since the two sets of assignments differ only in the intermediate variable used not in their resultant effect. Our technique accomplishes this by detecting equivalent variables and merging them accordingly. If `v` is merged into `u`, only two assignments remain and the desired reduction is achieved.

To exploit such opportunities we apply a variation of [20] to detect equivalent variables. The same algorithm has also been adapted by others, such as [3, 13, 22] for similar reasons.

$$\frac{visit(u := *v) \wedge v := w}{visit(u := *w)} \quad (25)$$

$$\frac{visit(u := *v) \wedge v := *w \wedge hold(w)}{add(u := *v) \wedge visit(v := *w)} \quad (26)$$

$$\frac{visit(u := *v) \wedge input(v)}{add(u := *v)} \quad (27)$$

$$\frac{visit(*u := v) \wedge u := w}{visit(*w := v)} \quad (28)$$

$$\frac{visit(*u := v) \wedge v := w}{visit(*u := w)} \quad (29)$$

$$\frac{visit(*u := v) \wedge v := \&w}{add(*u := v) \wedge visit(v := \&w)} \quad (30)$$

$$\frac{visit(*u := v) \wedge u := *w \wedge hold(w)}{add(*u := v) \wedge visit(u := *w)} \quad (31)$$

$$\frac{visit(*u := v) \wedge v := *w \wedge hold(w)}{add(*u := v) \wedge visit(v := *w)} \quad (32)$$

$$\frac{visit(*u := v) \wedge input(u) \wedge input(v)}{add(*u := v)} \quad (33)$$

Figure 7: Summary construction: deferral of dereference assignments.

6.2 Global assignment promotion

So far we have only discussed how to remove local variables from summaries. A final opportunity for compaction is by relocating assignments whose left-hand and right-hand sides are global variables. Such assignments are not specific to the context in which they are found and, therefore, they can be moved to the top-level summary without affecting correctness nor the result in any way. This avoids some unnecessary copying and propagation of global variables. Consider the following

```
R(p) with {} in {g := h}
```

where `g` and `h` are global variables. The assignment only involves global variables and is otherwise disjoint from `M`. Therefore it is not necessary to include `g1 := g2` into the summary for `M` as long as the assignment gets handled somewhere. It is best to move such assignments to the top-level so that they will be handled only once. A similar technique has been also proposed in [14].

7. EXPERIMENTAL RESULTS

We have extended the compaction algorithm described in the previous sections to work with the full C language, and implemented it as a component in the points-to analysis framework currently integrated in a full-scale compiler. For the primary experiments presented, the analysis parameters were chosen in order to closely match the algorithm described in this paper. This represents only one of many setups feasible using the general concepts we advocate. Preliminary results using exhaustive heap cloning are also presented because they provide some valuable insights into context sensitivity, though a detailed treatment is beyond the scope of this paper.

Call graph	Built on-the-fly
Recursion	Merge call graph cycles
Context	Sensitive (except call gph)
Constraints	Inclusion-based
Offset	Insensitive
Heap (“Analysis” Expr)	No Cloning
Heap (“Prelim HC” Expr)	Exhaustive Cloning
Flow	Insensitive

Figure 8: Key experimental parameters.

Figure 8 summarizes the major points-to analysis parameters used. The analysis is inclusion-based and the call graph is iteratively constructed. The analysis is context sensitive except that call graph itself is context insensitive. This means indirect calls made from a procedure are seen by all callers of that procedure. To match the derivation rules presented, both offset sensitivity and heap cloning were turned off for the results used in the primary discussions.

To demonstrate the usefulness of our approach, we evaluated the presented techniques on nineteen benchmarks from the SPEC92, SPEC95, and SPEC2000 suites. These benchmarks are all of SPEC's integer C programs excluding duplicates across suites (such as *022.li* and *130.li*). Some of the older benchmarks were chosen for comparison with previous works such as [14]. Figure 9 shows the lines-of-code, number of variables, and analysis run times for each benchmark. Times for runs without heap cloning are presented as “Analysis”, while the results for preliminary, heap cloning experiments are shown as “Prelim HC”. All of times have been rounded to the nearest non-zero second.

The run times include the loading of the intermediate representation of code from disk, the core analysis, and the storing of the results back to disk. The analyzer was compiled with debug information and no optimization, but was run on a 2.8 GHz Pentium 4 computer with 1G RAM.

The over-arching goal of this work is to facilitate a very precise yet scalable analysis framework. For this reason, all analysis times were limited to 40 minutes, as longer seemed intolerable for any user of an actual compiler. Most of the benchmarks completed in a few seconds, while *130.li*, *176.gcc*, and *254.gap* took substantially longer. Additionally, the benchmarks *132.ijpeg* and *253.perlbmk* did not finish within the allotted time.

For the five problematic benchmarks, the root cause of the extended analysis times was the effect of global variables on the compaction process. Because of their opaqueness, globals do not easily disappear from summaries as the bottom up process progresses. For this reason, purely global-to-global interactions are distilled out of the summaries. While this prevented the summary sizes from exploding, the transparency detection was still forced to make many local vari-

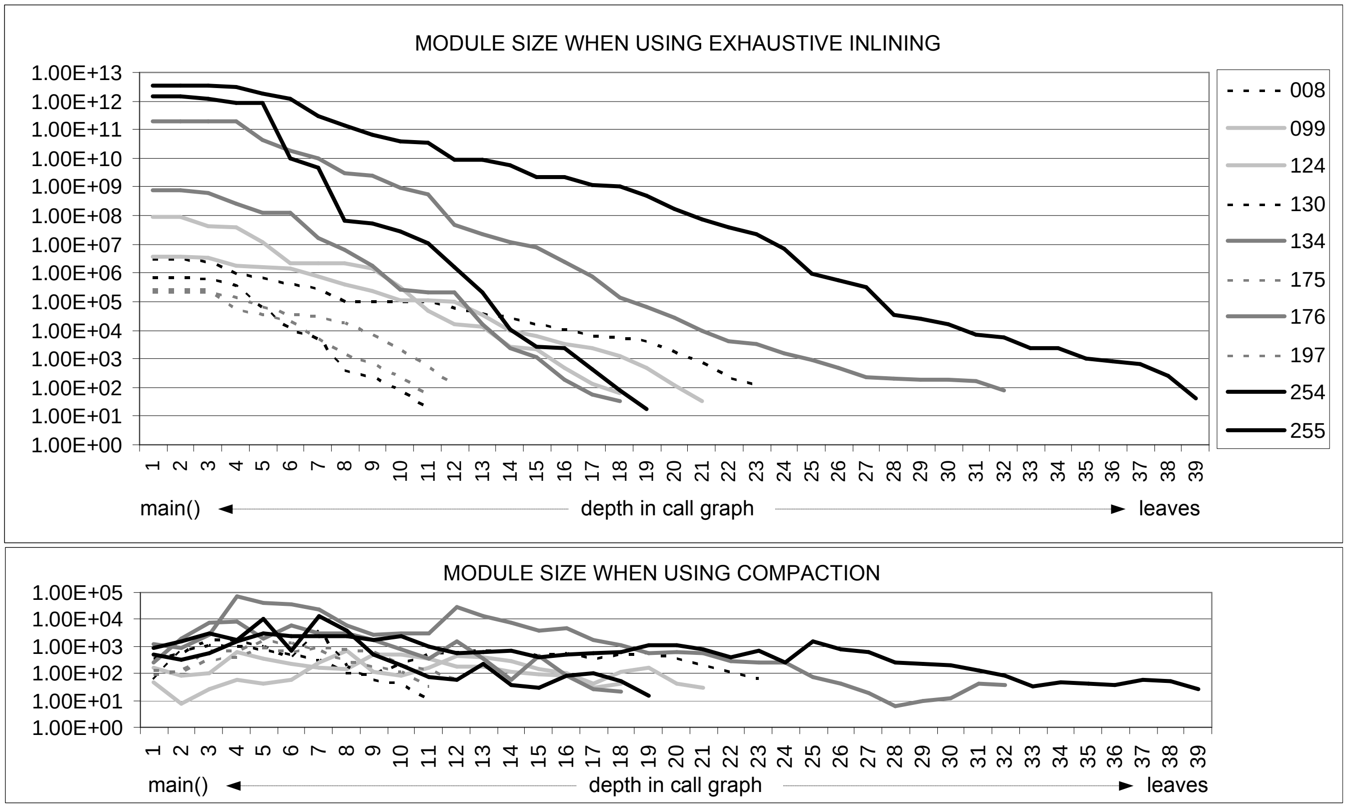

Figure 10: Log graph comparison of module size, in variables, versus call graph depth for exhaustive inlining and inlining using compaction.

ables opaque because of their interactions with global variables.

These opaque locals and the address assignments they generate are a direct cause of the disproportionate analysis times. For example, *130.li* is much smaller than *134.perl* yet consumes substantially more analysis time. Other issues noticeably affects the analysis times for a couple of benchmarks. In particular, *132.ijpeg* was impacted by the lack of offset sensitivity and heap cloning, grossly reducing the precision of its call graph, and the additional address assignments from opaque locals exacerbated an already address-assignment-burdened *253.perlbmk*.

In response to these results, we have extended our techniques to yield a more efficient, but equally precise treatment of global variables. *Our preliminary results show that this technique permits everything, including* 132.ijpeg *and* 253.perlbmk*, to run in under 20 minutes, but the details of this extension are beyond scope of this work.*

An exhaustive form of heap cloning was performed, and the preliminary results are shown in the "Prelim HC" column of Figure 9. As mentioned in Section 2.3, exhaustive heap cloning separates escaping objects along distinct call paths. For this reason, it can cause substantial increases in the analysis precision and the analysis problem size. Such an explosion in problem size is particularly the case for *008.espresso* which no longer completes and for *255.vortex* which has a four-fold increase in analysis time.

132.ijpeg is an important, but counterintuitive, example in which the addition of heap cloning actually decreased the total analysis time enabling it to quickly complete. *132.ijpeg* makes many indirect calls through heap allocated objects. These objects are allocated through a two-layer wrapper that itself is called indirectly through a heap object. The complete separation of the heap objects significantly improves the precision of the call graph, resulting in a much smaller program to analyze.

Reasonable comparisons between context-sensitive and insensitive analyses can be made using the size of the points-to graph as a metric. For the larger benchmarks, context sensitivity alone yields reductions from 10% (for benchmarks like *130.li*) to 30% (for those like *134.perl*, *176.gcc*, *254.gap*), to about 40% for *132.ijpeg*. A comparison against heap cloning is more difficult because the replication of heap locations can expand the points-to graph even though the actual precision is higher. However, the benefit of the increased precision to *132.ijpeg* is so significant that it is noticeable despite this difficultly. When compared to the context-insensitive result, the points-to graph for *132.ijpeg* is 80% smaller with heap cloning despite the heap object replication.

The major motivation for using compaction is to avoid the potentially explosive growth of summaries as modular points-to analysis progresses up the call graph. Figure 10 shows two

Benchmark	LOC	Vars	Analysis Time (s)	Prelim HC Time (s)
008.espresso	13505	7146	4	∞
023.eqntott	3393	944	1	1
099.go	28547	3184	1	1
124.m88ksim	17251	3130	2	2
129.compress	1426	148	1	1
130.li	6930	4422	313	∞
132.ijpeg	25897	4717	∞	70
134.perl	23969	10169	44	∞
164.gzip	7759	663	1	1
175.vpr	16973	4707	1	2
176.gcc	205747	56428	1700	∞
181.mcf	1909	408	1	1
186.crafty	18977	2457	1	1
197.parser	10924	3688	2	2
253.perlbmk	57541	26610	∞	∞
254.gap	59674	17808	1873	∞
255.vortex	52634	22134	46	200
256.bzip2	4637	427	1	1
300.twolf	19749	3401	2	5

Figure 9: Input size and analysis time across select SPEC92, SPEC95, and SPEC2000 integer benchmark suites (rounded to nearest non-zero second). The symbol ∞ is placed for the benchmarks which did not finish within the time limit.

graphs that plot the total number of variables in procedures at a given depth in the call graph. Procedure `main()` is always at a depth of one while the leaf procedures of a program span 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 100 thousand.

The upper graph plots the total number of variables when exhaustive inlining is used. The benchmarks are colored black, dark grey, light grey, dotted block, dotted dark grey. The colors divide the benchmarks in five pairs from the two largest to the two smallest benchmarks at the completion of the inlining. Clearly the growth is exponential meaning that some summarization mechanism would be needed to perform analysis for any but the smallest of benchmarks.

The lower graph plots the same result when compaction is performed between each inlining step. The benchmarks retain the coloring from the exhaustive graph. Instead of monotonically increasing as the process moves closer to main, the number of variables at a particular depth varies almost independently of depth, increasing and decreasing instead with respect to its ability to compact the summaries. The largest and smallest programs at a particular depth in the exhaustive graph are not necessarily the largest or smallest at that depth when compacted. The largest peaks in the compaction graph are generally points at which strongly connected components in the call graph forced the merger of a large number of procedures into one super-procedure. The results demonstrate that compaction is definitely effective at controlling summary growth.

8. CONCLUSIONS

This work presents compaction, an algorithm to reduce the size of summary information generated in the bottom-up process of modular, inclusion-based, context-sensitive points-to analysis. Specifically, this approach allows the replacement of original summaries with equivalent-yet-more-compact ones. The proposed method is based on the detection and elimination of transparent variables in the procedure summaries at each level of inclusion. The proposed methods effectively address the problems in reducing the size of summaries (i.e. aliasing, multi-level dereferences, redundancies, and global variable copying), and maintain the same level of analysis precision while effectively reducing the size of summaries.

The proposed methods have been implemented into a full-scale C compiler, and using empirical results, we have also shown that the proposed methods achieve a significantly greater reduction in summary size than previously published approaches. Our approach allow precise inclusion-based, context-sensitive points-to analysis to scale large, complex applications.

9. FUTURE WORK

The full form of our compaction approach was beyond the scope of this work. In particular, space prevented covering in detail two items closely tied to compaction:

- Compaction does not preclude offset sensitivity. While the offset sensitive compaction steps are more complex, the reduction in the resulting size, more than offsets the additional cost.
- The use of heap cloning in conjunction with compaction is largely seamless, but can be expensive if not carefully controlled, as we have demonstrated with preliminary results.

With regard to the framework as a whole, there are three major components of which compaction is one part of a bigger picture in terms of scalability:

- Experiments in Section 7 showed that interactions with global variables were a root problem for compaction because they are not easily removed and can readily make locals opaque. For this reason, we have developed another technique for completely isolating the global variables from the compaction mechanism.
- The top-down recovery process, introduced in Section 2, is necessary to complete the points-to information and can be made into a light-weight, powerful part of the process.

Future evaluation is needed to evaluate the interactions between these three critical mechanisms. Finally, a more concrete evaluation of the benefits of context sensitivity in general, as well as a better conceptual comparison of our work with other existing context-sensitive work would be informative.

REFERENCES

[1] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representation of polymorphic types with subtyping. In *Proceedings of Theoritical Aspects of Computer Software*, 1997.

[2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analysis for Elimination Unnecessary Synchronization from Java Programs. In *Proceedings of the Static Analysis Symposium*, 1999.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1988.

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

[5] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[6] Jess Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[7] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1999.

[8] Ben-Chung Cheng and Wen mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[9] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreenhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[10] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

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

[12] Manuel Fähndrich, Jefrrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[13] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[14] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the Static Analysis Symposium*, 2001.

[15] David Gay and Bjarne Steensgaard. Fast Escape Analysis and Stack Allocation for Java Programs. Technical report, Microsoft Research, 1998.

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

[17] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[18] Michael Hind and Anthony Pioli. Accessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the Static Analysis Symposium*, 1998.

[19] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of International Symposium on Software Testing and Analysis*, 2000.

[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 *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1997.

[22] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170, 2001.

[23] Jakob Rehof. Minimal typings in atomic subtyping. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1997.

[24] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorphic subtyping to cfl-reachability. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 2001.

[25] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[26] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[27] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1996.

[28] Zhendong Su, Manuel Fahndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 2000.

[29] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[30] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.