Faster Alias Set Analysis Using Summaries

Nomair A. Naeem and Ondřej Lhoták

University of Waterloo, Canada
{nanaeem,olhotak}@uwaterloo.ca

Abstract. Alias sets are an increasingly used abstraction in situations which require flow-sensitive tracking of objects through different points in time and the ability to perform strong updates on individual objects. The interprocedural and flow-sensitive nature of these analyses often make them difficult to scale. In this paper, we use two types of method summaries (callee and caller) to improve the performance of an interprocedural flow- and context-sensitive alias set analysis. We present callee method summaries and algorithms to compute them. The computed summaries contain sufficient escape and return value information to selectively replace flow-sensitive analysis of methods without affecting analysis precision. When efficiency is a bigger concern, we also use caller method summaries which provide conservative initial assumptions for pointer and aliasing relations at the start of a method. Using caller summaries in conjunction with callee summaries enables the alias set analysis to flow-sensitively analyze only methods containing points of interest thereby reducing running time. We present results from empirically evaluating the use of these summaries for the alias set analysis. Additionally, we also discuss precision results from a realistic client analysis for verifying temporal safety properties. The results show that although caller summaries theoretically reduce precision, empirically they do not. Furthermore, on average, using callee and caller summaries reduces the running time of the alias set analysis by 27% and 96%, respectively.

1 Introduction

Inferring properties of pointers created and manipulated by programs has been the subject of intense research [12, 24]. A large spectrum of pointer analyses, from efficient points-to analyses to highly precise shape analyses, have been developed. A useful tradeoff between the two extremes, and an increasingly used abstraction, is the alias set analysis. This static abstraction represents each runtime object with the set of all local pointers that point to it, and no others. The abstraction is neither a may-point-to nor a must-point-to approximation of runtime objects. Instead, each alias set represents exactly those pointers that reference the particular runtime object. As a result, like in a shape abstraction [26], every alias set (except the empty one) corresponds to at most one concrete object at any given point in time during program execution. This ability to statically pinpoint a runtime object enables strong updates which makes the abstraction suitable for analyses that track individual objects [6, 21, 10, 18]. We discuss the alias set analysis in more detail in Section 2.

Unlike a shape analysis which emphasizes the precise relationships between objects, and is expensive to model, an alias set analysis, like a pointer abstraction, focuses on local pointers to objects. This makes computing the alias set abstraction faster than

J. Knoop (Ed.): CC 2011, LNCS 6601, pp. 82–103, 2011.

shape analyses. However, since the analysis is flow-sensitive and inter-procedural it is still considerably slower than most points-to analyses. In this paper we propose two ways to further speed-up the alias set analysis; callee summaries providing effect and return value information and caller summaries that make conservative assumptions at method entry.

Flow sensitive analyses take into account the order of instructions in the program and compute a result for each program point. Although typically more precise than those that are insensitive to program flow, flow-sensitive analyses often have longer execution times than their insensitive counterparts. Computing such precise information for each program point is often overkill; clients of the analysis need precise results only at specific places. Long segments of code might exist where a client neither queries the analysis nor cares about its precision. As an example, consider a static verification tool that determines whether some property of lists and iterators is violated by the code in Figure 1. The verification tool is a client of the alias set analysis as it requires flow-sensitive tracking of individual objects to statically determine runtime objects involved in operations on lists and iterators. Notice that precise alias sets are required only when operations of interest occur. For the example, these are the two calls to `next` at lines 7 and 10 and the call to `add` at line 11. On the other hand, a typical alias set analysis computes flow-sensitive results for all program points irrespective of the fact that it is likely to be queried only at a few places.

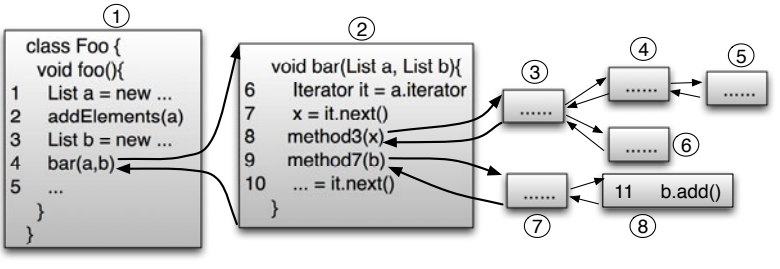

Fig. 1. Sample code illustrating the use of callee and caller summaries

In such situations, we propose the use of a selectively flow-sensitive alias set analysis that uses callee method summaries as a cheaper option. Only methods that contain a point of interest (which we call *shadows*), or transitively call methods containing shadows, are analyzed flow-sensitively. For all other methods, callee summaries providing effect information for the parameters of a method invocation and the possible return value are used. If callee summaries were available, only methods 1, 2, 7 and 8 from Figure 1 would have to be analyzed flow-sensitively since they contain shadows or call methods containing shadows. For the entire segment of code represented by methods 3-6, flow-sensitive information is not required and callee summaries can be used instead. In particular, while analyzing method 2 the alias set analysis need not propagate the analysis into method 3 at line 8 and instead its callee summary can be used. From the client's perspective this is acceptable since it does not query any program point within methods 3-6. In fact, as long as callee summaries contain sufficient information so that

foregoing flow-sensitive analysis of methods without shadows does not affect alias set precision in methods with shadows, the client's precision will be unaffected. Details of the construction of callee summaries and their use in the alias set analysis are given in Section 3.

The advantage any static analysis derives from interprocedurally analyzing a program is that the analysis need not make conservative worst case assumptions at method entry. This certainly holds true for the alias set analysis. At a callsite, the analysis ensures an appropriate mapping from the caller scope arguments to the callee scope parameters so that alias sets in the callee precisely represent aliasing at the start of the method. However, when efficiency is a bigger concern, we propose the use of caller summaries which are conservative and sound approximations of incoming alias sets. A direct benefit of using such summaries at method entries is that methods that were previously analyzed flow-sensitively only to obtain precise entry mappings for methods containing shadows no longer require flow-sensitive analysis. For example, since methods 1 and 7 in Figure 1 were analyzed flow-sensitively only because they contain calls to methods 2 and 8, with the added use of caller summaries this is no longer required. Only methods 2 and 8 will be analyzed flow-sensitively with caller summaries used to seed their initial alias sets and callee summaries used at all callsites.

Unlike callee summaries, caller summaries can affect the precision of the alias set abstraction since important aliasing information available at a particular callsite might not be propagated into the callee and instead some conservative assumption is made. The degree to which the use of caller summaries affects precision is dependent on the choice of caller summary as well as the client analysis.

This paper makes the following three contributions:

- We describe callee method summaries for the alias set analysis which provide sufficient information at a method callsite to forego flow-sensitive analysis of the callee without a loss of precision in the caller. We present algorithms to compute such summaries and a transfer function that employs the computed summary. (Section 3)
- We present the simplest caller summary as a proof of concept to using such summaries to flow-sensitively analyze even fewer methods. A transfer function for the alias set abstraction that uses both callee and caller summaries is also presented. (Section 4)
- We empirically evaluate the effect of caller summaries on the precision of a realistic client analysis and present precision metrics for the alias set abstraction. The effect on the running time of different incarnations of the alias set analysis is discussed. (Section 5)

2 Alias Set Analysis

The alias set abstraction employs abstract interpretation to summarize all possible runtime environments. The abstraction contains an alias set for every concrete object that could exist at run time at a given program point. The merge operation is a union of the sets of alias sets coming from different control flow paths. A given alias set $o^\sharp$ is exactly the set of local variables that point to the corresponding concrete object at run time. Individual alias sets do not represent may- or must- points-to approximations of

runtime objects, although the abstraction subsumes these relationships. If two pointers must point to the same object at a program point, then all alias sets in the abstraction for that point will either contain both pointers or neither. Similarly, if two pointers point to distinct objects at a program point then the abstraction at that point will not contain any alias sets containing both pointers.

2.1 Intermediate Representation and Control Flow Graph

We assume that the program has been converted into an SSA-based intermediate representation containing the following kinds of instructions:

$$s ::= Copy(x \leftarrow y) \mid Store(y.f \leftarrow x) \mid Load(x \leftarrow y.f) \mid$$
$$Null(x \leftarrow \textbf{null}) \mid New(x \leftarrow \textbf{new}) \mid Call(m(p_0 \cdots p_k))$$

The instructions copy pointers between variables, store and load objects to and from fields, assign null to variables, create new objects and call a method m. For method calls, the receiver is specified as the first argument p_0 followed by the arguments p_1 to p_k. ϕ instructions, introduced during SSA conversion, act as copy instructions with a different multi-variable copy for each incoming control flow edge.

The interprocedural control flow graph is created in the standard way; nodes represent instructions and edges specify predecessor and successor relationships. Each procedure begins with a unique $Start$ node and ends at a unique $Exit$ node. By construction, a call instruction is divided into two nodes; call and return. A call edge connects the call node in the caller with the start node in the callee. A return edge connects the exit node in the callee with the return node in the caller. A *CallFlow* edge connects a call node to its return node completely bypassing the callee (Figure 2). This edge is parameterized with the method it bypasses and the variable the return from the call is assigned to.

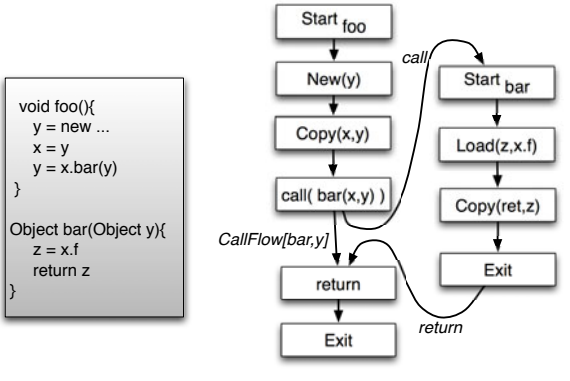

Fig. 2. Interprocedural control flow graph with *call*, *return* and *CallFlow* edges

2.2 Intra-procedural Alias Set Analysis

Flow-sensitivity enables the alias set analysis to precisely track abstract objects through different points in time. The analysis mimics the effect of program instructions in changing the targets of pointers and accordingly updates the alias sets representing each

runtime object. For example, consider the instruction x $\leftarrow$ **new**. At runtime an object o is allocated in the heap and x points to that object. Correspondingly, the static abstraction creates the alias set $\{\mathrm{x}\}$ representing the object's abstraction $o^\sharp$. Since a pointer can only point to one concrete object at a time, x points to the newly created object and none other. If a copy instruction $y \leftarrow x$ creates a new reference to the runtime object o the analysis mimics this effect by updating the alias set to $\{\mathrm{x,y}\}$. Hence, at all times each concrete object is represented by some alias set, though due to the conservative nature of the analysis there may be alias sets which represent no runtime object. For most instructions in the program, given an alias set representing some runtime object o, it is possible to compute the exact set of pointers which will point to o after the execution of the instruction.

An exception to this is the load from the heap ($v \leftarrow \mathbf{e}$). Since the abstraction only tracks local variables, the analysis is uncertain whether the object being loaded is represented by a given alias set $o^\sharp$ before the instruction, and whether the destination variable v should therefore be added to $o^\sharp$. To be conservative, the analysis accounts for both possibilities and creates two alias sets, one containing v ($o^\sharp \cup \{v\}$) and one not containing v ($o^\sharp \setminus \{v\}$). At this point a straightforward optimization can be applied; only objects that had previously escaped to the heap via a Store can be loaded. We have implemented this optimization in the alias set abstraction. At each program point, the abstraction computes two sets of alias-sets $\rho^\sharp$ and $h^\sharp$ with the condition that $h^\sharp \subseteq \rho^\sharp$ and that $h^\sharp$ contains only those alias sets which are abstractions of run time objects that have escaped into the heap.

In previous work [18] we presented the intra-procedural transfer functions for the alias set abstraction which we reproduce in Figure 3. The core of the transfer function is the helper function $[\![s]\!]_{o^\sharp}$ which, depending on the instruction, updates an existing

$$
\begin{aligned}
[\![s]\!]^1_{o^\sharp}(o^\sharp) &\triangleq \begin{cases} o^\sharp \cup \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ o^\sharp \setminus \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\sharp \\ o^\sharp \setminus \{v\} & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\} \\ o^\sharp & \text{if } s = \mathbf{e} \leftarrow v \\ \text{undefined} & \text{if } s = v \leftarrow \mathbf{e} \end{cases} \\
\mathit{focus}[h^\sharp](v, o^\sharp) &\triangleq \begin{cases} \{o^\sharp \setminus \{v\}\} & \text{if } o^\sharp \notin h^\sharp \\ \{o^\sharp \setminus \{v\}, o^\sharp \cup \{v\}\} & \text{if } o^\sharp \in h^\sharp \end{cases} \\
[\![s]\!]^1_{O^\sharp}[h^\sharp](O^\sharp) &\triangleq \begin{cases} \bigcup_{o^\sharp \in O^\sharp} [\![s]\!]^1_{o^\sharp}(o^\sharp) & \text{if } s \neq v \leftarrow \mathbf{e} \\ \bigcup_{o^\sharp \in O^\sharp} \mathit{focus}[h^\sharp](v, o^\sharp) & \text{if } s = v \leftarrow \mathbf{e} \end{cases} \\
[\![s]\!]^1_{\rho^\sharp}(\rho^\sharp, h^\sharp) &\triangleq [\![s]\!]^1_{\text{gen}} \cup [\![s]\!]^1_{O^\sharp}[h^\sharp](\rho^\sharp) \\
[\![s]\!]^1_{\text{gen}} &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = v \leftarrow \mathbf{new} \\ \emptyset & \text{otherwise} \end{cases} \\
[\![s]\!]^1_{h^\sharp}(\rho^\sharp, h^\sharp) &\triangleq [\![s]\!]_{O^\sharp}[h^\sharp] \left(\begin{cases} h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\} & \text{if } s = \mathbf{e} \leftarrow v \\ h^\sharp & \text{otherwise} \end{cases} \right) \\
[\![s]\!]^1_{\rho h^\sharp}(\rho^\sharp, h^\sharp) &\triangleq \left\langle [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp, h^\sharp), [\![s]\!]^1_{h^\sharp}(\rho^\sharp, h^\sharp) \right\rangle
\end{aligned}
$$

Fig. 3. Transfer functions on individual alias sets. The superscript^1 identifies the version of the transfer function; we will present modified versions of the transfer functions later in the paper.

alias set. For a copy instruction ($v_1 \leftarrow v_2$) any alias set that contains the source variable v_2 is modified by adding the target variable v_1, since after the instruction the source and target both point to the same location. Since a pointer can only point to one location at a time, instructions that overwrite a variable v modify an existing alias set by removing v as after the instruction v no longer points to the runtime object abstracted by this set. The store instruction ($\mathbf{e} \leftarrow v$) has no effect on an alias set since alias sets by definition only track local variables.

The *focus* operator in Figure 3 handles the uncertainty due to heap loads. As discussed earlier, only objects that were previously stored in the heap can be loaded. Therefore, for alias-sets not in $h^\sharp$, $focus(v, o^\sharp)$ removes v from $o^\sharp$ since the loaded object cannot possibly be represented by $o^\sharp$ and after the assignment v no longer points to $o^\sharp$. On the flip side, if $o^\sharp$ represents an escaped object, then it is split into two, one representing the single concrete object that may have been loaded, and the other representing all other objects previously represented by $o^\sharp$.

Two additional special cases are handled. First, for a store ($\mathbf{e} \leftarrow v$), all abstract objects that contain the variable v are added to $h^\sharp$. Second, for an allocation instruction, a new alias set containing only the destination variable v is created and added to $\rho^\sharp$.

Figure 4 graphically shows the effect of a sequence of three instructions on the alias set abstraction. For illustration we assume that before the first instruction, $\rho^\sharp$ and $h^\sharp$ already contain an alias set $\{\text{x,z}\}$ i.e. an abstraction of an object that is pointed to by local variables x and z and might also have external references from the heap. Note also the presence of a single empty alias set which represents all runtime objects that are not referenced through any local variables. This keeps the abstraction finite. With the allocation instruction, a new alias-set $\{\text{x}\}$ is added to $\rho^\sharp$. At the same time, $[\![s]\!]^\sharp_o$ removes x from the alias set $\{\text{x,z}\}$ since x no longer points to this runtime object. After the copy instruction both y and z point to the same runtime object. The heap load highlights a number of analysis features. First, note that the analysis determines that the loaded object cannot be the newly created object from instruction 1. Second, since $\{\text{x,z}\}$ was in $h^\sharp$, so is $\{\text{y,z}\}$. The analysis applies the *focus* operator. Third, notice the creation of the alias set $\{\text{z}\}$ which represents a loaded object that previously had no local variable references. Figure 4 illustrates the two key properties of alias set analysis (i) the abstraction can distinguish individual objects i.e. each alias set represents at most one runtime object and (ii) the transfer functions flow-sensitively track the effect of instructions on pointers. Each column represents what happens to a particular concrete

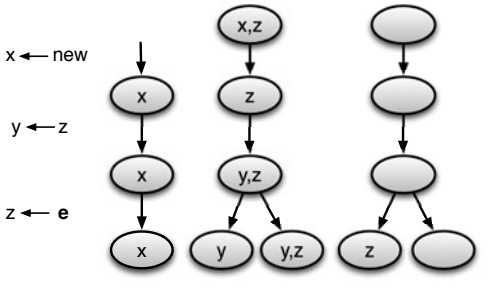

Fig. 4. An illustration of the transfer functions for computing the alias set abstraction

object as different instructions execute; all that changes is the set of pointers pointing to the object at different program points.

2.3 Inter-procedural Alias Set Analysis

The intra-procedural transfer function can be extended to be inter-procedural by defining the transfer functions for *call* and *return*. The overall effect of calling a function m is $[\![return]\!] \circ [\![m]\!] \circ [\![call]\!]$ for each possible callee. To determine the callees possible at each call site, we used a call graph computed using the default subset-based points-to analysis implemented in Spark [16]. The function $[\![call]\!]_{o^\sharp}$ is straightforward to define; actual arguments in each alias set are replaced by the corresponding parameters and all other variables are removed. Given a substitution r that maps each argument to its corresponding parameter, the function is defined in Figure 5.

$$[\![call]\!]^1_{o^\sharp}(o^\sharp) \triangleq \left\{ r(v) : v \in o^\sharp \cap dom(r) \right\}$$

$$rv(o^\sharp_c, o^\sharp_r) \triangleq \begin{cases} o^\sharp_c & \text{if } p \text{ does not return a value} \\ o^\sharp_c \cup \{v_t\} & v_s \in o^\sharp_r \\ o^\sharp_c \setminus \{v_t\} & v_s \notin o^\sharp_r \end{cases}$$

$$[\![return]\!]^1_{o^\sharp}(o^\sharp_c) \triangleq \{ rv(o^\sharp_c, o^\sharp_r) : o^\sharp_r \in [\![m]\!] \circ [\![call]\!] \}$$

Fig. 5. Transfer functions for $[\![call]\!]_{o^\sharp}$ and $[\![return]\!]_{o^\sharp}$

Defining the return from a function m is more challenging since any object that might be returned by m is abstracted by some alias set containing variables local to m. On its own this is insufficient to map variables from a callee alias set to one in the caller since it is unknown which caller variables, if any, pointed to the object before the call. Instead, the analysis uses a function that, given a call site and the computed flow function $[\![m]\!] \circ [\![call]\!]$, computes the appropriate caller-side alias sets after the function returns. For the $[\![return]\!]_{o^\sharp}$ function in Figure 5, $o^\sharp_c$ is the caller-side abstraction of an object existing before the call and the set $[\![m]\!] \circ [\![call]\!]$ contains all possible callee-side alias sets ($o^\sharp_r$) that could be returned. The function rv takes each such pair ($o^\sharp_c, o^\sharp_r$), where v_s is the callee variable being returned and v_t is the caller variable to which the return value is assigned. Intuitively, if the object that was represented by $o^\sharp_c$ in the caller before the call is returned from the callee (i.e. $v_s \in o^\sharp_r$), then v_t is added to $o^\sharp_c$. If some other object is returned, then v_t is removed from $o^\sharp_c$, since v_t gets overwritten by the return value. In the case of an object newly created within the callee, the empty set is substituted for $o^\sharp_c$, since no variables of the caller pointed to the object before the call. Overall, $[\![return]\!]$ yields the set of possible caller-side alias sets of the object after the call. We refer the interested reader to our previous work [20] for more details.

3 Callee Summaries

Although precise, the alias set analysis in its original form is expensive to compute. Using efficient data structures [19] and algorithms [22, 20] only improves the efficiency

to some extent. In situations where a faster running time is desired we propose the use of method summaries. In this section, we discuss the use of callee summaries that decrease the computation load, without any effect on a client analysis.

The key insight is that clients of a flow-sensitive whole program analysis often need precise information at a small subset of program points. On the other hand, a flow-sensitive program analysis computes precise information at all program points and therefore computes a lot more information than required. Computing this unnecessary information is wasteful and should be avoided. We use callee summaries to achieve this.

Before we explain the contents of a callee summary let us see how the alias set analysis can use such summaries. Consider a callsite, with a target method m. If an oracle predicts that a client of the alias set analysis never queries any program point within m or any methods transitively called by m, then computing flow-sensitive alias results for all methods in the transitive closure of m is unnecessary. Instead a callee summary, which provides information regarding the parameters and return value, could be used. For many client analyses such an oracle exists. In Section 5 we discuss one such client analysis that leverages alias sets in proving temporal properties of objects. The points of interest for this analysis i.e. the shadows, are operations that change the state an object is in and are statically known ahead of time. Additionally, callee summaries can be used for methods in the standard library; the alias set analysis can be seeded to use callee summaries for all chains of calls into the library. Analyses such as those detecting memory leaks and automatically deallocating objects [6, 21], that already use alias sets, could benefit from such summaries to only analyze application code.

The key requirement we put on a callee summary is that it should enable the analysis to bypass flow-sensitively analyzing a method without impacting precision in the caller. Table 1 provides a summary of the contents of such a summary. The summary is divided into escape (α_{esc}) and return value (α_{ret}) information.

Table 1. Callee Summary for a callsite with target method m

Escape Information (α_{esc})	
params	set of parameters (including receiver) that may be stored into the heap by m or procedures transitively called by m
Return Value Information (α_{ret})	
params	set of parameters (including receiver) that might be returned by m.
heap	might an object loaded from the heap be returned?
fresh	might a newly created object be returned?
escaped	might a newly created object be stored in the heap before being returned?
null	might a null reference be returned?

To determine the contents of a callee summary one must understand the effect of a method call on the alias set abstraction. First, the callee might escape the receiver or arguments of the call. This might occur *directly*, when a callee's parameter is stored in a field, or *indirectly*, when a parameter is copied to a local reference which is then stored. In Figure 6 the function `foo` escapes both its parameters, p directly via a store to

field f of class `Foo` and q indirectly by first copying the reference to y and then storing in $Foo.f$. Therefore, a callee summary analysis must track such copies and ultimately provide a list of all parameters that might have escaped.

Second, the return value from the callee might be assigned to a reference in the caller. To see how this might affect aliasing in the caller consider once again the example in Figure 6. The function `foo` returns the pointer y which is a copy of q, one of `foo's` parameters. Therefore, the returned reference is the argument which is mapped to q, in this case variable b. At run time, the effect of calling `foo` is that after the call, a and b must point to the same object. Let us examine the effect on the abstraction at the callsite if the interprocedural transfer functions from Figure 5 were used. $[\![call]\!]$ determines that b and q point to the same location and $[\![\texttt{foo}]\!]$ determines that q and y point to the same location. This leads $[\![return]\!]$ to infer that since b and y point to the same location and y is assigned to a, b and a must point to the same location after the call; an alias set containing both a and b is created in the caller. In order to forego flow-sensitive analysis of `foo` in favour of a callee summary, the summary must specify which of the callee's parameters might be returned so that similar updates can be made at the callsite. Other possible returned references include references to newly created objects or those loaded from the heap.

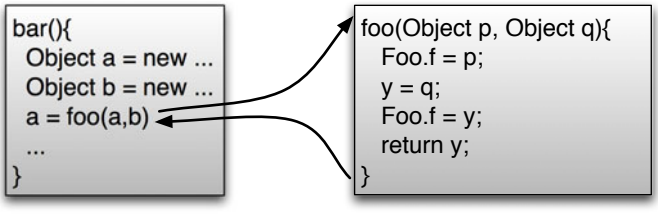

Fig. 6. An example illustrating the effect of a method call on alias sets in the caller

3.1 Computing Callee Summaries

The algorithm to compute the set of parameters that escape (α_{esc}) from a method m is presented in Figure 7. The algorithm takes as input a SSA-based control flow graph of the method and returns a set of indices which refer to the positions of parameters in the method's signature which might have escaped [1]. Lines 1-10 populate a worklist with variables that either escaped through a store or through a function call from within m. The algorithm then proceeds through each variable v in the worklist. Using the SSA property that each variable has a single reaching definition the algorithm retrieves the unique definition def of v (line 15). If def represents the Start node then v is a receiver or a parameter and the appropriate index is added to the mayEscape set. For a copy instruction $v \leftarrow s$, s is added to the worklist, since v and s both point to the same escaped object. Notice that the order between the instruction that escapes v and the copy from s to v does not matter, since in SSA-form once a variable is defined its value

[1] Recall from Section 2 that we write a function call as $m(p_0, \cdots p_k)$ where p_0 denotes the receiver of the call and p_1 to p_k are the arguments.

```
   input: SSA-based CFG of method m
   output:mayEscape
   declare mayEscape : Set[Int], WorkList: FIFOWorklist[Var], seen : Set[Var]
1  foreach instruction inst ∈ cfg do
2    switch inst
3      case inst = Store(v) : add v to WorkList end-case
4      case inst = CallSite(args, retval) :
5        foreach tgt ∈ callees(inst) do
6          WorkList += { args(i) : i ∈ EscapeSummaries(tgt) }
7        od
8      end-case
9    end-switch
10 od
11 while WorkList not empty
12   Select and Remove variable v from WorkList
13   if seen contains v then continue fi
14   add v to seen
15   def = uniqueDef(cfg,v)
16   switch def
17     case def = Start(p_0 ··· p_k): mayEscape += { i : p_i = v } end-case
18     case def = Copy(v,s): add s to WorkList end-case
19     case def = CallSite( args, retval ):
20       foreach tgt ∈ callees(def) do
21         WorkList += { args(i) : i ∈ RetValSummaries(tgt).params }
22       od
23     end-case
24     case def = Phi : foreach Copy(v,s) ∈ phi.defs(v) do add s to WorkList od end-case
25   end-switch
26 od
```

Fig. 7. Algorithm to compute callee escape summary (α_{esc}) for a method m

remains unchanged. If variable v is assigned the return value from a function call then all arguments corresponding to the parameters that might be returned are added to the worklist since these might have escaped (lines 19-23). A SSA ϕ instruction acts as a multi-variable copy statement.

Figure 8 presents the algorithm to compute the return value summary for a function m. The algorithm maintains a worklist of variables that might be returned. The worklist is seeded with the unique return variable of m. For each variable v in the worklist, depending on its unique definition, the return value summary and the worklist are updated. In lines 12-14, if v is defined at the `Start` node then, since a `Start` node defines the receiver or parameters of method m, the corresponding index of the parameter is stored in `params`. This represents the situation when the receiver or a parameter to m might be returned. Lines 15-23 update the return value summary if v is assigned the return value at a callsite. The return value summaries of all possible target methods at the callsite are consulted and the `fresh`, `heap` and `null` fields of the summary of m appropriately updated. If any of the return value summaries indicate that a receiver or parameter might be returned the corresponding argument is added to the worklist. $Copy$

```
   input: SSA-based CFG of method m
   output:retValSum
   declare WorkList: FIFOWorklist[Var], seen : Set[Var]
1  retValSum = RetValSum { params: Set[Int], heap = fresh = escaped = null = false }
4  if m.isVoid then return retValSum fi
5  Insert unique return variable into WorkList
6  while WorkList not empty
7    Select and remove variable v from WorkList
8    if seen contains v then continue fi
9    add v to seen
10   def = uniqueDef(cfg,v)
11   switch def
12     case def = Start(p_0 ··· p_k) :
13       retValSum.params += { i : p_i = v }
14     end-case
15     case def = CallSite( args, retval ) :
16       foreach tgt ∈ callees(def) do
17         calleeRetValSum = RetValSummaries(tgt)
18         if calleeRetValSum.fresh then retValSum.fresh = true fi
19         if calleeRetValSum.heap then retValSum.heap = true fi
20         if calleeRetValSum.null then retValSum.null = true fi
21         WorkList += { args(i) : i ∈ calleeRetValSum.params }
22       od
23     end-case
24     case def = Copy(v,s) : add s to WorkList end-case
25     case def = Phi :
26       foreach Copy(v,s) ∈ phi.defs(v) do add s to WorkList od
27     end-case
28     case def = Load : retValSum.heap = true end-case
29     case def = New : retValSum.fresh = true end-case
30     case def = Null : retValSum.null = true end-case
31   end-switch
32   foreach inst ∈ cfg do
33     switch inst
34       case inst = Store(v) :
35         if seen contains v then retValSum.escaped = true fi
36       end-case
37      case inst = CallSite (args, retval) :
38        foreach tgt ∈ callees(inst) do
39          if seen contains args(i) : i ∈ EscapeSummaries(tgt) then
40            retValSum.escaped = true
41          fi
42        od
43       end-case
44     end-switch
45   od
```

Fig. 8. Algorithm to compute the return value summary (α_{ret}) for a method m

and Phi instructions add sources of assignments to the worklist. $Load$, New and $Null$ instructions require an update to the corresponding `heap`, `fresh` and `null` fields of the return value summary. Two special cases must also be handled; if any possibly returned variable was stored in a field or escaped by a function called by m, `escaped` is set to true.

Since the callee summary of a function m depends on summaries of functions called by m, the algorithms presented must be wrapped in an interprocedural fixed-point computation. A worklist keeps track of all functions whose summaries may need to be recomputed. Whenever the summary of a function changes, all of its callers are added to the worklist. The computation iterates until the worklist becomes empty.

3.2 Using Callee Summaries

To leverage callee summaries in the alias set analysis the transfer functions from Figures 3 and 5 are modified. These modifications are presented in Figure 9. We denote the set of methods that contain shadows or transitively call methods containing shadows by M^*. The function $[\![call]\!]^1_{o^\sharp}$ is modified so that arguments in the caller are mapped to parameters in the callee only for methods in M^*, the methods that are still analyzed flow-sensitively. Since return instructions are only encountered in methods not using callee summaries no change is required to the return function $[\![return]\!]^1_{o^\sharp}$.

For methods not in M^*, we define the transfer function $[\![CallFlow]\!]$ for the similarly named edge connecting a call node to a return node in the caller. The $[\![CallFlow]\!]$ function uses two helper functions `mustReturn` and `mightReturn` which employ the return value summary α_{ret} to update alias sets by simulating the effect of analyzing the callee. The function `mustReturn` is true only when the object represented by $o^\sharp$ before the call is returned by the callee. Therefore the null, fresh and heap flags of the return value summary should be false since a non-null object, that was not allocated in the callee nor loaded from the heap should be returned. Additionally, $o^\sharp$ must contain the corresponding arguments of all parameters that might be returned by the callee. Parameters that might be returned are given by α_{ret}.params and the corresponding arguments are retrieved through the inverse function $\overline{r}$, where r is the function mapping arguments to parameters.

The helper function `mightReturn` determines whether $o^\sharp$ might be returned. This is true if $o^\sharp$ represents an escaped object ($o^\sharp \in h^\sharp$) and an object loaded from the heap might be returned. An object representing $o^\sharp$ might also be returned if at least one parameter, whose corresponding argument is in $o^\sharp$, might be returned. To handle the uncertainty when `mightReturn` is true, $[\![CallFlow]\!]$ accounts for both possibilities similarly to the *focus* operation. If the returned object must not be $o^\sharp$ then the variable assigned from the return of the callee cannot possibly point to $o^\sharp$ after the call.

The callee escape summary α_{esc} is utilized to update alias sets representing objects that might escape due to the function call (the function *escape* in Figure 9). If any of the corresponding arguments to parameters that escape (α_{esc}.params) are in an alias set in $\rho^\sharp$, the alias set is added to $h^\sharp$ since the function call escapes the parameter. The transfer function must also handle situations when the callee allocates and returns a new object. If α_{ret}.fresh is true and the return from the callee is assigned to variable v, an alias set

$$[\![call]\!]^2_{o^\sharp}(o^\sharp) \triangleq \begin{cases} [\![call]\!]^1_{o^\sharp}(o^\sharp) & \text{if } s = call \wedge target(call) \in M^* \\ \emptyset & \text{otherwise} \end{cases}$$

$$mustReturn(o^\sharp, \alpha_{ret}) \triangleq \begin{cases} true & \text{if } !\alpha_{ret}.\text{fresh} \wedge !\alpha_{ret}.\text{heap} \wedge !\alpha_{ret}.\text{null} \wedge \\ & \quad \forall p, p \in o^\sharp : p \in \overline{r}(\alpha_{ret}.\text{params}) \\ false & otherwise \end{cases}$$

$$mightReturn[h^\sharp](o^\sharp, \alpha_{ret}) \triangleq \begin{cases} true & \text{if } (o^\sharp \in h^\sharp \wedge \alpha_{ret}.\text{heap}) \vee \\ & \quad \exists p, p \in o^\sharp : p \in \overline{r}(\alpha_{ret}.\text{params}) \\ false & otherwise \end{cases}$$

$$[\![CallFlow]\!]^2_{o^\sharp}[h^\sharp, m, v](o^\sharp) \triangleq \begin{cases} \emptyset & \text{if } m \in M^* \\ \{o^\sharp \cup v\} & \text{if } m \notin M^* \wedge mustReturn(o^\sharp, \text{m}.\alpha_{ret}) \\ \{o^\sharp \setminus v, o^\sharp \cup v\} & \text{if } m \notin M^* \wedge mightReturn[h^\sharp](o^\sharp, \text{m}.\alpha_{ret}) \\ \{o^\sharp \setminus v\} & \text{otherwise} \end{cases}$$

$$[\![s]\!]^2_{\text{ret}} \triangleq \begin{cases} \{\{v\}\} & \text{if } s = CallFlow[m, v] \wedge m \notin M^* \wedge \text{m}.\alpha_{ret}.fresh \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![s]\!]^2_{O^\sharp}[h^\sharp](O^\sharp) \triangleq \begin{cases} \bigcup_{o^\sharp \in O^\sharp} [\![s]\!]^1_{o^\sharp}(o^\sharp) & \\ \qquad \text{if } s \notin \{v \leftarrow e, CallFlow[m, v]\} & \\ \bigcup_{o^\sharp \in O^\sharp} [\![CallFlow]\!]^2_{o^\sharp}[h^\sharp, m, v](o^\sharp) & \text{if } s = CallFlow[m, v] \\ \bigcup_{o^\sharp \in O^\sharp} focus[h^\sharp](v, o^\sharp) & \text{if } s = v \leftarrow e \end{cases}$$

$$[\![s]\!]^2_{\rho^\sharp}(\rho^\sharp, h^\sharp) \triangleq [\![s]\!]^1_{\text{gen}} \cup [\![s]\!]^2_{\text{ret}} \cup [\![s]\!]^1_{O^\sharp}[h^\sharp](\rho^\sharp)$$

$$escape(\rho^\sharp, h^\sharp, m) \triangleq h^\sharp \cup \{o^\sharp \in \rho^\sharp : \exists p, p \in o^\sharp : p \in \overline{r}(m.\alpha_{esc}.\text{params})\}$$

$$[\![s]\!]^2_{\text{esc}}(m) \triangleq \begin{cases} \{\{v\}\} & \text{if } \text{m}.\alpha_{ret}.fresh \wedge \text{m}.\alpha_{ret}.escaped \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![s]\!]^2_{h^\sharp}(\rho^\sharp, h^\sharp) \triangleq \begin{cases} [\![s]\!]^2_{O^\sharp}[h^\sharp]\left(h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\}\right) & \text{if } s = e \leftarrow v \\ [\![s]\!]^2_{\text{esc}}(m) \cup [\![s]\!]^2_{O^\sharp}[h^\sharp] escape(\rho^\sharp, h^\sharp, m) & \\ \qquad \text{if } s = CallFlow[m, v] \wedge m \notin M^* & \\ [\![s]\!]^2_{O^\sharp}[h^\sharp]\left(h^\sharp\right) & \text{otherwise} \end{cases}$$

Fig. 9. Modified transfer functions for the alias set analysis using callee summaries

containing only v is added to $\rho^\sharp$. If the freshly created object might have been stored in the heap before being returned (α_{ret}.escaped is true) a similar alias set is added to $h^\sharp$.

4 Caller Summaries

In this section we present caller summaries as a mechanism to speed up the interprocedural context-sensitive alias set analysis. Although static analyses that infer properties of pointers can be useful even when the analysis is carried out locally on individual methods, such analyses shine most when computed interprocedurally. The added ability to carry forward computed pointer and aliasing information from a caller into a callee by mapping arguments to parameters can significantly improve precision. However, when efficiency is a bigger concern a natural trade-off is to forego some precision by conservatively assuming initial pointer and aliasing relationships for the parameters of a method.

The reason using caller summaries improves efficiency is that it decreases the number of the methods the must be analyzed flow-sensitively. Let us revisit the example in

Figure 1. Using callee summaries enables the alias set analysis to discard flow-sensitive analysis of methods 3-6 since they do not contain any shadows. However, even though methods 1 and 7 do not contain shadows, they are analyzed flow-sensitively to ensure that at a callsite to a method containing a shadow, precise information can be mapped into the callee. In an analysis that uses caller summaries to make conservative assumptions at every method entry, flow-sensitively analyzing such methods is un-needed since the precise information computed at the callsite will never be propagated into the callee.

We have implemented a conservative mechanism for computing caller summaries. In our summaries, initial alias sets are created for the parameters of a method such that the abstraction at the start of the method specifies that any two parameters might be aliased. In our intermediate representation the method `bar` in the example from Figure 1 has three parameters; the `this` receiver and the two `List` references a and b. The caller summary for this method contains the following sets: $\{\}$, $\{$`this`$\}$, $\{a\}$, $\{b\}$, $\{$`this`,$a\}$, $\{$`this`,$b\}$, $\{a,b\}$ and $\{$`this`,$a,b\}$. Notice that given these alias sets the only conclusion that can be drawn is that the three parameters might be aliased i.e. no must or must-not relationships exist between the parameters. This is overly conservative. Firstly, the caller summary does not take into account any type information. Although a and b are both references to a `List` data structure, the receiver `this` is of type `Foo` and, unless `Foo` is declared a supertype of `List`, a reference of type `Foo` can never point to a `List` object. Secondly, the caller summaries do not leverage any pointer information. For example, subset-based points-to analyses that use allocation sites as their object abstraction are often performed at onset for constructing a callgraph. Using this type of pointer analysis could potentially improve the precision of the caller summary in situations where the pointer analysis can specify that parameters a and b were created at different allocation sites.

Our reason for using a naive caller summary was to investigate the maximum precision degradation due to such summaries. Whereas the callee summaries presented in the preceding section do not affect precision, caller summaries do. As an example let us look more closely at the example in Figure 1. The method `bar` receives two `List` references, a and b. An alias set analysis which does not utilize caller summaries is able to differentiate between the two references. In particular, at the start of method `bar` the analysis infers that a and b must-not alias (two separate lists were created at lines 1 and 3 and assigned to a and b respectively, and a reference of one is never copied to the other). However, the naive caller summary assumes that a and b could be aliased. Hence the precision of the alias set analysis degrades i.e. fewer must-not facts are computed.

This decrease in precision can cascade into client analyses. For example suppose a client of the alias set analysis is a verification tool for the property that an iterator's underlying list structure has not been modified when its `next` method is invoked (executing such code results in a runtime exception). If caller summaries are not used, the analysis infers that the iterator's underlying list i.e the list referenced by a, is never modified since a and b must-not point to the same object and the code only modifies the list referenced by b. Hence, the client analysis can prove that line 10 is not a violation of the property. However, when caller summaries are used, the client analysis infers that the list pointed to by reference a might be modified (the caller summary suggests that a and b might be aliased and an element is added at line 11 to the list pointed to by

reference b). Hence the client analysis loses precision since it can no longer prove that the `next` operation at line 10 is safe w.r.t. the property being verified. We empirically evaluate the loss in precision of using caller summaries on the alias set analysis and a client analysis in Section 5.

$$[\![call]\!]^3_{o^\sharp}(o^\sharp) \triangleq callerSummaries(target(call))$$

$$[\![CallFlow]\!]^3_{o^\sharp}[h^\sharp, m, v](o^\sharp) \triangleq \begin{cases} \{o^\sharp \cup v\} & \text{if } mustReturn(o^\sharp, \text{m}.\alpha_{ret}) \\ \{o^\sharp \setminus v, o^\sharp \cup v\} & \text{if } mightReturn[h^\sharp](o^\sharp, \text{m}.\alpha_{ret}) \\ \{o^\sharp \setminus v\} & \text{otherwise} \end{cases}$$

$$[\![s]\!]^3_{h^\sharp}(\rho^\sharp, h^\sharp) \triangleq \begin{cases} [\![s]\!]^2_{O^\sharp}[h^\sharp]\left(h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\}\right) & \text{if } s = e \leftarrow v \\ [\![s]\!]^2_{\text{esc}}(m) \cup [\![s]\!]^2_{O^\sharp}[h^\sharp] escape(\rho^\sharp, h^\sharp, m) & \text{if } s = CallFlow[m, v] \\ [\![s]\!]^2_{O^\sharp}[h^\sharp]\left(h^\sharp\right) & \text{otherwise} \end{cases}$$

Fig. 10. Updated transfer functions for the alias set analysis using callee and caller summaries

Modifying the alias set analysis to use caller summaries is straightforward. Figure 10 shows those transfer functions which have been modified from their earlier version (Figure 9). First, the *call* function is modified. Instead of mapping arguments to parameters, the caller summary provides the set of alias sets to seed the callee's analysis. Second, callee summaries are used for all methods instead of only those not in M^*.

5 Experiments

Any standard dataflow analysis framework can be used to compute the alias set abstraction using the original transfer functions from Section 2 or the subsequently modified versions from Sections 3 and 4. We have chosen to implement these incarnations as instances of the efficient interprocedural finite distributive subset (IFDS) algorithm of Reps et al. [22]. The algorithm requires an analysis domain $\mathcal{P}(D)$ for some finite set D, and transfer functions that are distributive. The alias set abstraction satisfies these conditions where D is the set of all possible alias sets. The key to IFDS's efficiency is the distributivity of transfer functions which enables it to evaluate the functions on individual alias sets, rather than on the entire set of alias sets at a program point. Space restrictions limit us in providing interesting details of the algorithm. Instead we refer the interested reader to the ever growing body of work discussed in Section 6.

For the experiments we used the DaCapo Benchmark suite, version 2006-10-MR2 with the standard library from JDK 1.3.1_12 for antlr, pmd and bloat, and JDK 1.4.2_11 for the rest, since they use features not present in JDK 1.3. The intermediate representation is constructed using the Soot Framework [29] with reflective class loading modelled through reflection summaries obtained using ProBe [15] and *J [9]. To give an indication of the size of these benchmarks we computed the number of methods statically reachable in the control flow graph created by Soot and present these in Table 2. Time taken to pre-compute the callee summaries using the algorithms discussed in Section 3 are also shown.

In our experiments we have used a static analysis that verifies conformance to temporal properties specified using a statemachine-based specification [2] as a client of the

Table 2. Number of statically reachable methods and the time to precompute callee summaries

Benchmark	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
Reachable Methods	4452	5955	14912	27408	11418	14437	7358	7821	9365	14961
Callee SummaryTime(s)	8	12	29	72	28	50	10	10	15	29

alias set abstraction. In previous work [18], we presented a two stage approach to verifying such properties. In the first stage an alias set abstraction of objects in the program is computed. The second stage uses this abstraction to compute an abstraction for the state an object, or group of objects, is in. This enables statically verifying whether the state machine might end up in an error state indicating a violation of the property.

Our choice of client analysis was dictated by two reasons. First, each temporal property specifies its own points of interest; only events that transition the state machine of that property are considered shadows. By choosing different properties we ensure a varying set M^*, the set of methods for which callee summaries are used. The properties we experimented with are presented in Table 3. The code fragment from Figure 1 uses the FSI property. Shadows for FSI are the next operation on an iterator and updates on the `Collection` type e.g. add, clear, remove. A second reason for choosing this client analysis is that it cleanly teases apart the computation of the alias set abstraction and its use in computing the state abstraction. This enables us to measure the precision and efficiency of the alias set abstraction in a real-world scenario.

Table 3. Temporal properties investigated to obtain a varying set M^*

FailSafeEnum (FSE): A vector should not be updated while enumerating it
FailSafeEnumHash(FSEH): A hashtable should not be updated while enumerating its keys/values
FailSafeIter (FSI): A collection should not be updated while iterating over it
HasNext (HN) The hasNext method should be called prior to every call to next on an iterator
HasNextElem (HNE): The hasNextElem method should be called prior to every call to nextElement on an enumeration
Reader (R): A Reader should not be used after its InputStream has been closed
Writer (W): A Writer should not be used after its OutputStream has been closed

We call a pair containing a benchmark and temporal property a *test case*. Since not all benchmarks exercise all temporary properties we have chosen to present results only for test cases when a temporal property is applicable for a benchmark e.g. the antlr benchmark never uses a Writer and hence the corresponding temporal property is inapplicable.

5.1 Shadow Statistics

Section 3 proposed the use of callee summaries for methods not in M^* and Section 4 proposed the use of caller summaries for all methods thereby requiring flow-sensitive analysis of only methods containing shadows (S). We measured the percentage of reachable methods that are in M^* and S and present these in Table 4. The maximum percentage of methods in M^* was for the test case jython-FSI where 59.9% of the methods are in M^*. Notice that the methods containing shadows for jython-FSI are only 0.6%

Table 4. Percentage of reachable methods that contain shadows or transitively call methods with shadows (M^*) and methods that contain shadows (S)

	antlr		bloat		chart		fop		hsqldb		jython		luindex		lusearch		pmd		xalan	
	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S
FSE	56.3	0.7					47.6	0.1	1.6	0.1	59.8	0.4	1.6	0.4	1.1	0.2	9.5	0.1	50.0	0.6
FSEH	56.3	1.1							2.8	0.1	59.8	0.4	1.1	0.3	0.8	0.2			49.9	0.3
FSI			56.3	4.2	50.6	0.6	47.7	0.5	54.0	0.1	59.9	0.6	53.1	0.4	52.4	0.6	52.4	1.0	50.0	0.5
HN			56.0	2.6	50.5	0.4	47.6	0.1	54.0	0.1	59.8	0.2	53.1	0.2	52.3	0.3	52.2	0.5	0.1	0.1
HNE	56.3	0.7	0.1	0.1					1.6	0.1	59.8	0.2	0.5	0.2	0.3	0.1	6.6	0.1	49.9	0.2
R	7.5	0.2							4.2	0.3	59.8	0.2	0.9	0.1	2.3	0.3	7.7	0.1	49.9	0.1
W			55.8	0.5			47.6	0.3	5.7	0.6			0.8	0.1	1.7	0.3	0.2	0.1	49.9	0.4

indicating that most methods are in M^* since they call methods containing shadows. On average (geometric mean) M^* contains 11.9% of the reachable methods implying that callee summaries are used for the remaining 88.1%. When using both callee and caller summaries a mere 0.3% of reachable methods (average of set S) require flow-sensitive analysis.

5.2 Efficiency

To measure the effect of summaries on the time required to compute the alias set abstraction we computed the abstractions using the three versions of the transfer functions. In Table 5 we show the running time of the original alias set abstraction (ORIG), the alias set abstraction which uses only callee summaries (CS) and the abstraction using both callee and caller summaries (CCS). The times for CS and CCS include the time for computing the callee summary and CCS also includes the time to compute the caller summary.

For all test cases, the time required to compute the abstraction is reduced when callee summaries are used for methods not in M^*. The greatest reduction is for pmd-writer which takes 99.6% less time to compute (6670 vs 29 seconds). The reason for this is quite obvious; for pmd-writer, M^* contains only 12 methods out of the 9365 reachable methods. On average the use of callee summaries reduces the time to compute the alias set abstraction by 27%. Introducing caller summaries has a more significant impact; an average reduction of 96% is witnessed over the entire test set.

5.3 Client Analysis Precision

As discussed earlier we chose a static analysis that verifies conformance to temporal properties as a client of the alias set analysis. The analysis represents temporal properties as state machines where operations on object(s) transition the state machine associated with the object(s). To distinguish the objects on which operations are performed, the analysis uses an alias set abstraction. The result of the analysis is a list of shadows that cannot be verified by the analysis; these include actual violations and false positives.

To evaluate the effect of using caller summaries on the precision of the analysis (callee summaries have no effect on precision) we executed the client analysis using

Table 5. Time taken to compute the alias set abstraction for the original transfer functions (ORIG), the transfer functions leveraging Callee Summaries (CS) and the transfer functions employing both Callee and Caller Summaries (CCS)

		antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
FSE	ORIG	484			5934	1351	2390	1017	580	2638	5052
	CS	349			5422	220	1524	118	151	37	4484
	CCS	15			108	40	71	19	18	26	112
FSEH	ORIG	500				1426	2020	1054	576		6395
	CS	386				226	1397	120	133		5834
	CCS	18				39	77	17	17		51
FSI	ORIG		1810	1653	4057	1316	2335	1057	553	3685	5100
	CS		1683	1022	4051	651	1671	391	407	2069	5099
	CCS		563	72	109	37	69	17	17	29	119
HN	ORIG		1601	1665	3735	1406	2225	1019	485	5273	5262
	CS		1556	1035	3289	714	1390	393	388	5031	164
	CCS		455	44	115	41	70	18	17	29	45
HNE	ORIG	457	1607			1450	2233	1098	562	5182	4361
	CS	358	119			220	1481	137	121	32	3588
	CCS	16	18			40	77	18	17	26	44
R	ORIG	511				1416	2205	1097	563	4348	3280
	CS	55				238	1487	123	123	35	3172
	CCS	13				39	73	16	16	27	48
W	ORIG		1551		3840	1450		1067	607	6670	3468
	CS		1411		3553	318		120	146	29	3323
	CCS		19		447	37		17	18	28	66

the ORIG and CCS abstractions. As per our discussion in Section 4, we expected a decrease in precision since caller summaries cause the alias set abstraction to compute fewer aliasing facts. However, the results surprised us; none of the 54 test cases showed any degradation in the client analysis. The CCS abstraction contained sufficient must and must-not aliasing at each shadow of a test case to produce the same transitions on the abstract state machine.

Our conclusion from this experiment is that even though caller summaries cause a theoretical decrease in precision, this does not automatically translate into precision loss for the client analysis. Situations exist where the benefits of using caller summaries heavily outweigh the slight chance of losing precision.

5.4 Fine-Grained Precision Metrics

When the client analysis did not show a loss of precision, we set out to develop a fine-grained metric for evaluating precision of alias sets. Using the alias set abstraction we compute must and must-not alias pairs for variables live at the shadows of each test case. Then we sum the alias pairs for all shadows in a test case to give us two precision metrics: MA the aggregated must-alias pairs and MNA the aggregated must-not alias pairs. As expected, the metric values for ORIG and CS are identical indicating that no precision is lost by using callee summaries. Table 6 presents the results of ORIG

(alternately CS) vs CCS[2]. For each test case the MA and MNA values for ORIG are presented. Below this is a number indicating the number of alias pairs that are lost with CCS. For example, the MA value for jython-FSE is 51 indicating that 51 different alias-pairs were identified at the shadows of this test case. The absence of a number below indicates no decrease in precision when using caller summaries. The MNA value for jython-FSE is 189. The -7 below indicates that 7 must-not alias pairs were lost when caller summaries were used.

Table 6. Alias set abstraction precision in terms of aggregated must aliasing (MA) and must not aliasing (MNA) metrics computed at the shadows for each test case

	bloat		chart		jython		luindex		lusearch		pmd		xalan	
	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA
FSE					51	189 -7	6	98 -2	18	91 -1	0	35 -1	371	1180 -64
FSEH					930 -21	1546 -19	4	63 -4	1	17			179	384 -2
FSI	1152	18858 -611	3344 -212	6244 -254	404	583 -32	76	226 -1	77	233 -1	459 -3	1505 -79	350	1042 -22
HN	606	7584 -328	704 -14	1529 -214	322	386	95	202	58	112 -1	127	839 -53	0	13
HNE	0	21 0			11	133 -10	8	91 -2	0	13	0	41	45	194 -1
R					253 -9	427 -14	59 -44	80	203 -44	222	56	130 -7	671	1395 -20
W	7	306 -55					56 -41	83	200 -41	219	0	22	524	799 -21

Of the 54 test cases, only 9 showed a degradation in the MA precision metric. The four highest degradations were for luindex-R (75%), luindex-W (73%), lusearch-R (22%) and lusearch-W (21%). The average (geometric mean) degradation for the 9 test cases was 8%. 31 of the 54 test cases also noted a decrease in the MNA metric. The maximum decrease was 17% for bloat-W with an average decrease of 4%.

6 Related Work

Kildall's framework [14] for intraprocedural dataflow analysis was extended by Sharir and Pnueli [27] to perform context-sensitive interprocedural dataflow analysis using either the *call-strings* or *functional* approach. The functional approach computes the effect of each procedure by composing functions for individual instructions in the procedure thereby obtaining a summary function $f_p : \mathcal{D} \rightarrow \mathcal{D}$, where $\mathcal{D}$ is the dataflow analysis domain. Once the summary function for a procedure has been computed it is used at each call site of the procedure to model the effect of the call. Sagiv, Reps and Horwitz [22] extended the original formalism to $\mathcal{P}(D)$ for a finite set D with the

[2] Due to space limitations we do not present results for antlr, fop and hsqldb.

condition that the functions on individual dataflow facts should be distributive. Distributivity of transfer functions enables the graphical representation of these functions as bipartite graphs with $O(D)$ nodes. The IFDS algorithm has been used to solve both locally separable problems such as reaching definitions, available expressions and live variables, and non-locally-separable problems such as uninitialized variables and copy-constant propagation. Its efficiency makes it suitable for computing a variety of useful static abstractions [10, 23, 28, 31, 13, 25, 19].

Other frameworks for computing procedure summaries have also been proposed. Gulwani and Tiwari [11] developed procedure summaries in the form of constraints that must be satisfied for some generic assertion to hold at the end of the procedure. Their key insight was to use weakest preconditions of such generic assertions. Furthermore, for efficiency they use strengthening and simplification of these preconditions to ensure early termination. The approach has been used to compute two useful abstractions; unary uninterpreted functions and linear arithmetic. Recently, Yorsh et al. [32] introduced an algorithm which also computes weakest preconditions and relies on simplification for termination. They describe a class of complex abstract domains (including the class of problems solvable using IFDS) for which they can generate concise and precise procedure summaries. Their approach uses symbolic composition of the transfer functions for the instructions in the program to obtain a compact representation for the possibly infinite calling contexts.

In contrast to the related work discussed above, we propose a technique to reduce the number of methods that must be analyzed using any of the approaches discussed above (our implementation uses the IFDS algorithm [22] to compute the alias set abstraction). Under certain conditions, instead of computing expensive procedure summaries through IFDS, our analysis uses cheaper callee summaries without a loss of precision.

Cherem and Rugina [7] present a flow-insensitive, unification-based context-sensitive analysis to construct method summaries that describe heap effects. The analysis is parameterized for specifying the depth of the heap to analyze (k) and the number of fields to track per object (b). Varying the values for k and b results in different method summaries; smaller values produce lightweight summaries whereas larger values result in increased precision. Method summaries were shown to significantly improve a client analysis that infers uniqueness of variables i.e. when a variable holds the only reference to an object.

Also related are analyses which traverse the program callgraph (mostly bottom-up but some top-down analyses have also been proposed) and compute a summary function for each procedure [30, 4, 5]. This summary function is then used when analyzing the callers.

Escape analysis has been widely studied [8, 3, 1, 30] and used in a variety of applications ranging from allocating objects on the stack to eliminating unnecessary synchronization in Java programs. To determine whether an object can be allocated on the stack and whether it is accessed by a single thread, Choi et al. [8] compute object escape information using *connected graphs*. A connected graph summarizes a method and helps identify non-escaping objects in different calling contexts. In their work on inferring aliasing and encapsulation properties for Java [17], Ma and Foster present a static analysis for demand-driven predicate inference. Their analysis computes predicates such as checking for uniqueness of pointers (only reference to an object), parameters that are lent (callee does not change uniqueness) and those that do not escape a callee.

7 Summary

This paper presented callee and caller summaries as a means to improve the efficiency of an alias set analysis. We described the information required from a callee summary to ensure that their use does not decrease precision at a callsite. Algorithms to compute the callee summary and the alias set transfer function leveraging the summaries were also presented. Through experimental evidence we showed that a client analysis and alias set precision metrics are unaffected by the use of callee summaries. On average a 27% reduction in the running time to compute the abstraction was witnessed.

In situations where some loss of precision is acceptable in favour of larger gains in efficiency, we showed how caller summaries that make assumptions about pointer and aliasing relationships at method entry can be employed. In order to gauge the maximum decrease in precision, we chose to use a conservative caller summary which assumes that any two parameters of a method might be aliased. Empirical evaluation of the effect of using caller summaries on the precision of the client analysis revealed no decrease in the abilities of the client analysis. For a fine-grained evaluation of precision, two metrics deriving aggregated must and must-not aliasing between variables were calculated. The average decrease was 8% for the must- and 4% for the must-not alias metric. The running time for computing the alias set abstraction decreases by 96% on average if both callee and caller summaries are used.

Future directions include experimenting with other client analyses of the alias set abstraction, using callee and caller summaries for the standard library, and developing less conservative caller summaries such as those briefly mentioned in Section 4.

Acknowledgements. This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada and Ontario Ministry of Research and Innovation.

References

1. Aldrich, J., Chambers, C., Sirer, E.G., Eggers, S.J.: Static analyses for eliminating unnecessary synchronization from Java programs. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 19–38. Springer, Heidelberg (1999)
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005, pp. 345–364 (2005)
3. Blanchet, B.: Escape analysis for object-oriented languages: application to Java. In: OOPSLA 1999, pp. 20–34 (1999)
4. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL 1999, pp. 133–146 (1999)
5. Cheng, B.-C., Hwu, W.-M.W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In: PLDI 2000, pp. 57–69 (2000)
6. Cherem, S., Rugina, R.: Compile-time deallocation of individual objects. In: ISMM 2006, pp. 138–149 (2006)
7. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: Adsul, B., Vetta, A. (eds.) CC 2007. LNCS, vol. 4420, pp. 172–186. Springer, Heidelberg (2007)
8. Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: OOPSLA 1999, pp. 1–19 (1999)
9. Dufour, B.: Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University (June 2004)

10. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. 17(2), 1–34 (2008)
11. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)
12. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: PASTE 2001, pp. 54–61 (2001)
13. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: SIGSOFT FSE 1995, pp. 104–115 (1995)
14. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206 (1973)
15. Lhoták, O.: Comparing call graphs. In: PASTE 2007, pp. 37–42 (2007)
16. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
17. Ma, K.-K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: OOPSLA 2007, pp. 423–440 (2007)
18. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: OOPSLA 2008, pp. 347–366 (2008)
19. Naeem, N.A., Lhoták, O.: Efficient alias set analysis using SSA form. In: ISMM 2009, pp. 79–88 (2009)
20. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the IFDS algorithm. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 124–144. Springer, Heidelberg (2010)
21. Orlovich, M., Rugina, R.: Memory leak analysis by contradiction. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 405–424. Springer, Heidelberg (2006)
22. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61 (1995)
23. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 284–302. Springer, Heidelberg (2005)
24. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
25. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science 167(1-2), 131–170 (1996)
26. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Trans. Program. Lang. Syst. 20(1), 1–50 (1998)
27. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, ch. 7, pp. 189–233. Prentice-Hall, Englewood Cliffs (1981)
28. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: ISSTA 2007, pp. 174–184 (2007)
29. Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java bytecode using the soot framework: Is it feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
30. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA 1999, pp. 187–206 (1999)
31. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
32. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: POPL 2008, pp. 221–234 (2008)