Bootstrapping: A Technique for Scalable Flow and Context-Sensitive Pointer Alias Analysis

Vineet Kahlon

NEC Labs America, Princeton, NJ 08540, USA.

vineetkahlon@gmail.com

Abstract

We propose a framework for improving both the scalability as well as the accuracy of pointer alias analysis, irrespective of its flow or context-sensitivities, by leveraging a three-pronged strategy that effectively combines *divide and conquer, parallelization* and *function summarization*. A key step in our approach is to first identify small subsets of pointers such that the problem of computing aliases of any pointer can be reduced to computing them in these small subsets instead of the entire program. In order to identify these subsets, we first apply a series of increasingly accurate but highly scalable (context and flow-insensitive) alias analyses in a cascaded fashion such that each analysis A_i works on the subsets generated by the previous one A_{i-1}. Restricting the application of A_i to subsets generated by A_{i-1}, instead of the entire program, improves it scalability, i.e., A_i is *bootstrapped* by A_{i-1}. Once these small subsets have been computed, in order to make our overall analysis accurate, we employ our new summarization-based flow and context-sensitive alias analysis. The small size of each subset offsets the higher computational complexity of the context-sensitive analysis. An important feature of our framework is that the analysis for each of the subsets can be carried out independently of others thereby allowing us to leverage parallelization further improving scalability.

1. Introduction

Static analysis has recently emerged as a powerful technique for detecting potential bugs in large-scale real-life programs. To be effective, such static analyses must satisfy two key conflicting criteria - accuracy and scalability. Since static analyses typically work on heavily abstracted versions of the given program, they may potentially generate many false positives. The key challenge, therefore, is to reduce the number of false positives while keeping the analysis scalable. However the accuracy and scalability of most static error detection methods strongly hinge on the precision and efficiency of the underlying pointer analysis, especially for C programs. This makes an accurate as well as scalable pointer analysis indispensable for such applications.

We propose a framework for improving both the scalability as well as the accuracy of pointer alias analysis, irrespective of its flow or context-sensitivities, by leveraging a combination of *divide and conquer*, *parallelization* and *function summarization*. The key strategy underlying our analysis is to first use an efficient and scalable analysis to identify small subsets of pointers, called *clusters*, that have the crucial property that the computation of the aliases of a pointer in a program can be reduced to the computation of its aliases in each of the small clusters in which it appears. This, in effect, decomposes the pointer analysis problem into much smaller sub-problems where instead of carrying out the pointer analysis for all the pointers in the program, it suffices to carry out separate pointer analyses for each small cluster. Furthermore, given a cluster only statements that could potentially modify aliases of pointers in that cluster need be considered. Thus each cluster induces a (usually small) subset of statements of the given program to which the pointer analysis can be restricted thereby greatly enhancing its scalability. Once this partitioning has been accomplished, a highly accurate pointer analysis can then be leveraged. The small size of each cluster then offsets the higher computational complexity of this more precise analysis.

In order to identify the clusters, we apply a series of increasingly accurate (but less scalable) alias analyses in a cascaded fashion such that each analysis A_i works on the pointer subsets generated by the previous one A_{i-1} and not on the entire program. Restricting the application of A_i to subsets generated by A_{i-1}, instead of the entire program, improves its scalability. In other words, A_i is *bootstrapped* by A_{i-1}. Once these small clusters have been computed, in order to make our overall analysis accurate, we employ our new summarization-based flow and context-sensitive alias analysis. We start the bootstrapping by applying the highly scalable Steensgaard's analysis [13] to identify clusters as points-to sets defined by the (Steensgaard) points-to graph. Since Steensgaard's analysis is bidirectional, it turns out that these clusters are, in fact, equivalence classes of pointers and so the resulting clusters are referred to as *Steensgaard Partitions*. In case there exist Steensgaard partitions whose cardinality is too large for a context-sensitive alias analysis to be viable (as determined by a threshold size), Andersen's analysis [1] is then performed separately on these large partitions. Thus whereas Andersen's analysis, which is more accurate than Steensgaard's, might have been less scalable on the original program, leveraging Steensgaard's analysis to first partition the set of pointers in the program improves it scalability, viz., Steensgaard's analysis bootstraps Andersen's analysis. Indeed, the maximum Steensgaard partition size that we encountered in our benchmark suite was 596 for the *sendmail* program that had a total of 65134 pointers thus clearly demonstrating the reduction in the scale of the problem. Note that Andersen's points-to analysis, being unidirectional, is more precise than Steensgaard's which is bidirectional. Hence it produces smaller clusters than Steensgaard's analysis. For instance, the Steensgaard partition of size 596 in the *sendmail* example was broken up into several Andersen clusters, the maximum size of which was 193. Usually, Andersen clusters are small enough so that our summary-based flow and context-sensitive pointer analysis becomes viable.

Figure 1 shows that the plot of frequency of each cluster size in the Linux Driver *autofs* which is typical across all the examples that we considered. The white squares represent Steensgaard partitions whereas black squares represent Andersen clusters. Note (i) the high density of both white and black squares for low values of cluster size, (ii) the stark difference in maximum size of Steensgaard partitions (isolated white square to the far right) and Andersen clus-

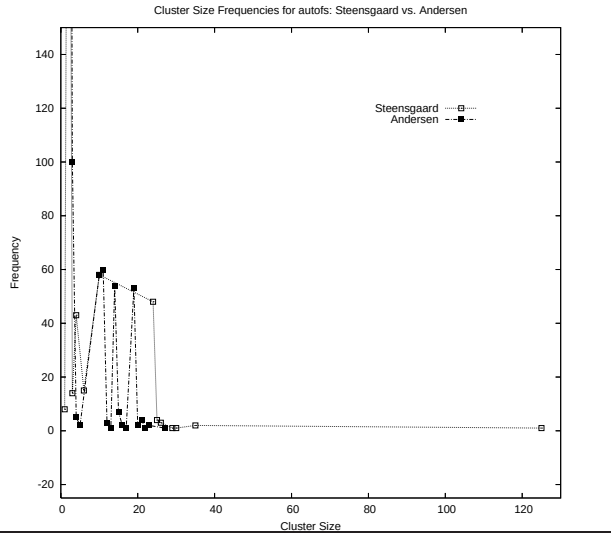

Figure 1. Cluster Size Frequencies : Steensgaard vs. Andersen

ters. It shows that by simply generating Steensgaard/Andersen clusters, we can *localize* the aliasing problem to only the relevant pointers making it possible to efficiently compute precise aliases for them. More generally, bootstrapping allows one to string together a series of pointer analyses of increasing accuracy till the subsets are small enough to ensure scalability of a highly precise alias analysis, context-sensitive or not. This not only ensures scalability but also improves the accuracy of the overall analysis. Furthermore, since these clusters can be analyzed independently of each other, it gives us the ability to leverage parallelization which is important given the increasing prevalence of multi-core processors.

Another key contribution of this paper is a new summarization-based approach for flow and context-sensitive alias analysis. One of the key bottlenecks in context-sensitive alias analysis is that the number of contexts grows exponentially with the number of functions in the given program. Typically, large-scale C programs tend to have a large number of small functions which can easily cause the number of contexts to blow up and overwhelm the analysis.

Function summarization offers an effective solution for handling this blow-up in the number of contexts. Indeed, summarization is, by its very nature, compositional thus making it robustly scalable to large code. A summarization-based technique relying on the building of partial transfer functions has been explored before for pointer analysis [18]. The key idea there was to capture all the different ways in which the points-to relation between the parameters of a function can be modified by executing the function body. Our summarization technique is different and relies on capturing sequences of assignments to pointers that are relevant for aliasing. Summarization of assignment sequences, which are simpler objects than points-to graphs, can be accomplished more compactly. In order to compute summaries for a given Andersen cluster A, we first use a dataflow analysis to identify the pointers V_A and the set St_A of statements modifying these pointers that may affect aliases of pointers in A. Next, the summary tuples are computed by analyzing pointers in V_A. Recursion is handled by processing the strongly connected components of the function call graph in reverse topological order. For each strongly connected component, we use a fixpoint algorithm to compute summary tuples for each function in that component. The aliases of a pointer in a given context can then be computed by splicing together local assignment sequences as captured by the summary tuples for all the functions in the order in which they appear in the given context. Tracking sequences of assignments locally within each function and splicing them together across functions in the given context makes our analysis flow-sensitive not just locally inside a function, as was the case in [15], but for the entire program.

A crucial point is that bootstrapping allows us to exploit locality of reference. Indeed, since Andersen's clusters are typically small, by restricting summary computation to each individual cluster ensures that the resulting summaries will also be small. Secondly, the number of statements modifying values of pointers in a given cluster also tend to be few and highly localized to a few functions. This in turn, obviates the need for computing summaries for functions that don't modify any pointers in the given cluster which typically accounts for majority of the functions. Note that without clustering it would be hard to ensure viability of the summarization approach. Thus it is the synergy between divide and conquer and summarization (and parallelization) that ensures scalability of our approach.

Another advantage of bootstrapping is flexibility as it gives us the ability to pick and choose which clusters to explore. Indeed, based on the application, we may not be interested in accurate aliases for all pointers in the program but only a small subset. As an example, for lockset computation used in data race detection, we need to compute must-aliases only for lock pointers. Thus we need to consider only clusters having at least one lock pointer. In fact, as is to be expected, since a lock pointer can alias only to another lock pointer, we need to consider clusters comprised solely of lock pointers. This makes our analysis extremely flexible as it can be adapted on-the-fly based on the demands on the application. Moreover one may choose to engage different pointer analysis methods to analyze different clusters based on their sizes and access densities resulting in a hybrid approach. Thus bootstrapping makes our framework flexible which can adapted according to the needs of the target application.

Related Work. Most scalable pointer alias analyses for C programs have been context or flow-insensitive. Steensgaard [13] was the first to propose a unification based highly scalable flow and context-insensitive pointer analysis. The unification based approach was later extended to give a more accurate one-flow analysis that has one-level of inclusion constraints [3, 4] and bridges the precision gulf between Steensgaard's and Andersen's analysis [1]. Inclusion-based algorithms have been explored to push the scalability limits of alias analysis [2, 9, 10, 11, 14]. For many applications where flow-sensitivity is not important, context-sensitive but flow-insensitive alias analyses have been explored [6, 7].

The idea of partitioning the set of pointers in the given program into clusters and performing an alias analysis separately on each individual cluster has been explored before [19]. However, the clustering in [19] was based on treating pointers, references or dereferences thereof, purely as syntactic objects and by computing a transitive closure over them with respect to the equality relation. A clustering based on Steensgaard's analysis takes into account not just assignments between pointers (at the same level in Steensgaard's hierarchy) but also points-to relation between objects (at different levels in the hierarchy). As a result, Steensgaard partitions are much more refined, i.e., smaller in size than the ones based on purely syntactic criteria. Furthermore, cascading of several analyses for increasing precision via cluster refinement has, to the best of our knowledge, not been considered before.

There is also substantial prior work on flow and context-sensitive alias analysis. An early attempt, shown to handle up to 4KLOC, was proposed in [12]. The use of partial transfer functions for summarization [18] discussed before has been shown to handle code up to 20KLOC of C code. A method for computing aliasing information for stack allocated data structures was presented in [5]. A compositional and interprocedural pointer analysis for Java programs was given in [16]. By using symbolic data structures like

BDDs to represent summaries encoded in terms of transfer functions, context and flow-sensitive alias analysis has been shown to scale up to 25KLOC [21]. The use of BDDs for pointer analysis was pioneered by Zhu [20, 21]. In related work, it has been shown that BDDs can be used for efficiently computing context-sensitive inclusion based points-to sets for Java programs [2]. More recently, the use of BDDs to represents relations in Datalog [17] has been proposed. Representing pointer analysis as a logic programming problem allows it be formulated as a sets of datalog rules which can then be used to compute BDDs for a context-sensitive alias analysis [15] with limited flow sensitivity. This approach has been shown to be successful for Java but less so for C programs. A flow and context-sensitive alias analysis with (limited) path sensitivity but one which requires user annotations and is therefore semi-automatic, was presented in [8]. In contrast, our analysis is fully automatic.

To sum up, our key new contributions are a framework for scalable flow and context-sensitive pointer alias analysis that

- ensures scalability as well as accuracy by applying a series of analysis in a cascaded fashion
- is flexible
- is fully automatic
- provides a new summarization technique that is more succinct than existing ones.

2. Bootstrapping

We start by fixing some notation. For a given program $Prog$, let P denote the set of all pointers of $Prog$. Then for $Q \subseteq P$, we use St_Q to denote the set of statements of *Prog* executing which may affect the aliases of some pointer in Q. Furthermore, for $q \in Q$, $Alias(q, St_Q)$ denotes the set of aliases of q in a program $Prog_Q$ resulting from *Prog* where each assignment statement not in St_Q is replaced by a *skip* statement and all conditional statements of $Prog$ are treated as evaluating to *true*. In other words, all statements of *Prog* other than those in St_Q are ignored in $Prog_Q$.

Our main goal in this section is to show how to compute subsets $P_1, ..., P_m$ of P such that

(i) $P = \bigcup_i P_i$

(ii) For each $p \in P$, $Alias(p, St_P) = \bigcup_i Alias(p, St_{P_i})$

(iii) The maximum cardinality of P_i (and of St_{P_i}) over all i is small. This is required in order to ensure scalability in computing the sets $Alias(p, St_{P_i})$.

Note that goal (ii) allows us to decompose computation of aliases for each pointer $p \in P$ in the given program to computing aliases of p with respect to each of the subsets P_i in the program $Prog_{P_i}$. This enables us to leverage divide and conquer. However, in order to accomplish this decomposition care must be taken in constructing the sets $P_1, ..., P_n$ which need to be defined in a way so as not to miss any aliases. We refer to sets $P_1, ..., P_m$ satisfying conditions (i) and (ii) above as a *Disjunctive Alias Cover*. Furthermore, if the sets P_1,...,P_m are all pairwise disjoint then they are referred to as a *Disjoint Alias Cover*.

Remark 1. As is usual (see, for example, [3]), we assume, for the sake of simplicity, that each pointer assignment in the given program is of one of the following four types (i) $x = y$, (ii) $x = \&y$, (iii) $*x = y$, and (iv) $x = *y$. These cases capture the main issues in pointer alias analysis. The general case can be handled with minor modifications to our analysis. Recursion is allowed. Heaps are handled by representing a memory allocation at program location loc by a statement of the form $p = \&alloc_{loc}$. A memory deallocation is replaced by a statement of the form $p = NULL$. We flatten all structures by replacing them with collections of separate variables - one for each field. This converts all accesses to fields of structures into regular assignments between such variables. While this was required in our framework for model checking programs, an important side benefit is that this makes our pointer analysis field sensitive. Pointer arithmetic is, for now, handled in a naive manner, by aliasing all pointer operands with the resulting pointer. Function pointers are handled as in [5].

```
main(){
  1a: p = &a;
  2a: q = &b;
  3a: r = &c;
  4a: q = p;
  5a: q = r;
}
```

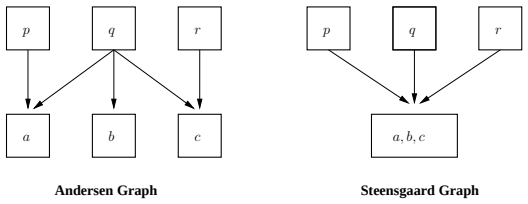

Figure 2. Steensgaard vs. Andersen Points-to Graphs

Remark 2. In the interest of brevity, we touch on (the now standard) Steensgaard's analysis and associated terminology like points-to relations, Steensgaard points-to graph, etc., only briefly without providing a formal description which can be found in [13].

2.1 Steensgaard Partitioning

In Steensgaard's analysis [13], aliasing information is maintained as a relation over abstract memory locations. Every location l is associated with a label or set of symbols ϕ and holds some content α which is an abstract pointer value. Points-to information between abstract pointers is stored as a points-to graph which is a directed graph whose nodes represent sets of objects and edges encode the points-to relations between them. Intuitively, an edge $e : v_1 \rightarrow v_2$ from nodes v_1 to v_2 represents the fact that a symbol in v_1 may point to some symbol in the set represented by v_2. The effect of an assignment from pointers y to x is to equate the contents of the location associated with y to x. This is carried out via unification of the locations pointed-to by y and x into one unique location and if necessary propagating the unification to their successors in the points-to graph. Assignments involving referencing or dereferencing of pointers are handled similarly. Since Steensgaard's analysis does not take the directionality of assignments into account, it is bidirectional. This makes it less precise but highly scalable. Figure 2 shows the Steensgaard points-to graph for a small example.

The Steensgaard Points-to Hierarchy. The key feature of Steensgaard's analysis that we are interested in is the well known fact that the points-to sets so generated are equivalence classes. Hence these sets define a partitioning of the set of all pointers in the program into disjoint subsets that *respect the aliasing relation*, i.e., a pointer can only be aliased to pointers within its own partition. We shall henceforth refer to each equivalence class of pointers generated by Steensgaard's analysis as a *Steensgaard Partition*. For pointer p, let n_p denote the node in the Steensgaard points-to graph representing the Steensgaard partition containing p. A Steensgaard points-to graph defines an ordering on the pointers in P which we refer to as the *Steensgaard points-to hierarchy*. For pointers $p, q \in Q$, we say that p is *higher than* q in the Steensgaard points-to hierarchy, denoted by $p > q$, or equivalently by $q < p$, if n_p and n_q are distinct nodes and there is a path from n_p to n_q in the Steensgaard points-to

```
main(){
  int a, b;
  int *x;
  1a: x = &a;
  2a: y = &b;
  3a: p = x;
  4a: *x = *y;
}
```

Figure 3. Identifying relevant statements

graph. Also, we write $p \sim q$ to mean that p and q both belong to the same Steensgaard partition. The *Steensgaard depth* of a pointer p is the length of the longest path in the Steensgaard points-to graph leading to node n_p. That the notion of Steensgaard depth is well-defined follows from the fact that a Steensgaard points-to graph is a forest of directed acyclic graphs (see comment below).

Important Remark. The Steensgaard points-to graph should not be confused with graph of the points-to relation. The graph of the points-to relation can contain cycles. However, a Steensgaard points-to graph which is over sets (equivalence classes) of pointers and not individual pointers is *always* acyclic. Consider the assignment $*p = p$ which creates a loop in the graph of the points-to relation. Since both $*p$ and p belong to the same Steensgaard equivalence class ($p \sim *p$), they will be represented by the same node in the Steensgaard points-to graph. Since the Steensgaard points-to graph only has edges between different nodes, we can deduce that it will be acyclic for the above statement. This ensures that the $<$ relation introduced above is well-defined. Note that such cycles in the points-to graph can arise in common situations involving cyclic data structures, void pointers, etc. We therefore distinguish between the points-to hierarchy and the points-to relation. Henceforth whenever we use the term points-to hierarchy we mean the Steensgaard points-to hierarchy.

Divide and Conquer. When computing the aliases of pointers in a given Steensgaard partition P, we want to restrict our analysis only to those statements of $Prog$ that may affect aliases of pointers in P. A standard, but important, observation is that aliases of a pointer in P can be affected only by assignments to either a pointer in P or a pointer q higher in the Steensgaard points-to hierarchy than some pointer in P. Assume now that our goal is to compute Andersen aliases of pointers in P. Then it suffices to restrict our analysis only to statements that directly modify values of pointers in the set $P_{\geq}$ comprised of all pointers q such that either $q > p$ or $q \sim p$. However, we show that we can do more effective slicing.

Consider the program $Prog$ shown in fig. 3. The Steensgaard partitions for program $Prog$ are $\{a, b\}, \{y\}$ and $\{p, x\}$. Suppose that we are interested in the aliases of pointers in the partition $P = \{a, b\}$. Note that since x is (one level) higher in the points-to hierarchy than both a and b, any modification to $*x$ or x may also affect aliases of a or b. Thus any assignment to $*x$ or x needs to be taken into consideration while performing the alias analysis for a and b. Hence we need to add `4a` and `1a` to St_P, the set of statements of $Prog$ that need be considered while computing aliases of pointers in P. Note that `1a` affects aliases of a due to the fact that x points-to a. We also need to take into account assignments between pointers at the same depth in the points-to hierarchy. Indeed, from statement `4a` we can deduce that the value of $*x$ depends on the value of $*y$ and so any statement that modifies $*y$ or y (as in this program $y > *y$) should also be taken into account thereby including `2a` into St_P. Then it is easily seen that no more statements need be added to St_P. Observe that if we had simply taken St_P to be the set of all statements modifying any pointer in $P_{\geq}$, we would also have included statement `3a` which does not affect the aliases of a or b. In a large program, the effect of discarding such statements can be quite significant.

The above example points to a simple fixpoint computation (algorithm 1) that, given a Steensgaard partition P, can identify the set of statements St_P that may affect aliases of pointers in P. The procedure first computes the set of variables (or references or dereferences thereof) V_P which may affect aliases of pointers in P. We start by initializing V_P to P (line 2). Next we iteratively compute new pointers that can affect aliases of pointers in V_P. The values of a pointer $p \in V_P$ can be modified either via a direct assignment to p (line 5) or via an assignment to a dereferencing of a pointer q where either $q > p$ or (the case of a cyclic points-to relation) q, $*q$ and p all occur in the same Steensgaard partition (line 8). When all such pointers have been discovered, we simply return the set of all statements that modify a pointer in V_P and hence the possible aliases of a pointer in P.

Algorithm 1 Computing Relevant Statements

1: **Input:** Program $Prog$, Steensgaard Partition: P.
2: Initialize V_P to P.
3: **repeat**
4: $\quad V'_P = \emptyset$.
5: $\quad$ **if** there exists a pointer q such that there exists a statement of the form $p = q$, where $p \in V_P$ and $q \notin V_P$ **then**
6: $\quad\quad$ add q to V'_P.
7: $\quad$ **end if**
8: $\quad$ **if** there exists a pointer q such that either (i) $q > p$, or (ii) $q \sim *q$ and $*q \sim p$ (cyclic case), where $p \in V_P$, and there exists a statement of the form $*q = r$ with $r \notin V_p$ **then**
9: $\quad\quad$ add $*q, q, r$ to V'_P.
10: $\quad$ **end if**
11: $\quad V_P = V_P \cup V'_P$.
12: **until** $V'_P = \emptyset$
13: **return** the set of all statements of $Prog$ of the form $p = q$, where $p \in V_P$.

Soundness. The key step is showing that given a Steensgaard partition P, restricting Andersen's analysis to statements in St_P does not result in the loss of any aliases. Towards that end, we first give a necessary and sufficient condition for when two pointers are aliased to each other. We start with some definitions. A pointer p is said to be *semantically equivalent* to q at location l if p and q have the same value at l (even if they are syntactically different).

Definition 3 (Complete Update Sequence). *Let $\lambda : l_0, ..., l_m$ be a sequence of successive program locations and let π be the sequence $l_{i_1} : p_1 = a_0$, $l_{i_2} : p_2 = a_1$,..., $l_{i_k} : p_k = a_{k-1}$, of pointer assignments occurring along λ. Then π is called a complete update sequence from p to q leading from locations l_0 to l_m iff*

- *a_0 and p_k are semantically equivalent to p and q at locations l_0 and l_m, respectively.*
- *for each j, a_j is semantically equivalent to p_j at l_{i_j},*
- *for each j, there does not exist any (semantic) assignment to pointer a_j between locations l_{i_j} and $l_{i_{j+1}}$, to a_0 between l_0 and l_{i_1} and to p_k between l_{i_k} and l_m along λ.*

A related concept is that of maximally complete update sequences.

Definition 4 (Maximally Complete Update Sequence). *Given a sequence $\lambda : l_0, ..., l_m$ of successive control locations starting at the entry control location l_0 of the given program, the maximally complete update sequence for q leading from locations l_0 to l_m along λ is the complete update sequence π of maximum length, over all pointers p, from p to q (leading from locations l_0 to l_m)*

```
main(){
  int *a, *b, *c;
  int **x, **y;
  1a: b = c;
  2a: x = &a;
  3a: y = &b;
  4a: *x = b;
}
```

Figure 4. Complete vs. Maximally Complete Update Sequences

occurring along λ. If π is an update sequence from p to q leading from locations l_0 to l_m, we also call it a maximally complete update sequence from p to q leading from locations l_0 to l_m.

Typically l_1 and l_m (see above definitions) are clear from the context. Then we simply refer to π as a complete or maximally complete update sequence from p to q. As an example, consider the program in figure 4. The sequence `4a` is a complete update sequence from b to a leading from `1a` to `4a`, but not a maximally complete one. It can be seen that `1a, 4a` is a maximal completion of `4a`. Note that at location `4a`, $*x$ is not syntactically but semantically equal to a due to the assignment at location `2a`. Maximally complete update sequences can be used to characterize aliasing.

Theorem 5. *Pointers p and q are aliased at control location l iff there exists a sequence λ of successive control locations starting at the entry location l' of the given program and ending at l such that there exists a pointer a with the property that there exist maximally complete update sequences from a to both p and q (leading from l' to l) along λ.*

We can now get back to our original goal of showing that given a Steensgaard partition P, restricting pointer analysis to program statements in St_P does not cause us to miss any aliases.

Theorem 6. *Given a set of pointers Q belonging to the same Steensgaard partition and pointer $p \in Q$, $Alias(p, St_P) = Alias(p, St_Q)$, where P is the set of pointers of the given program.*

Computing Andersen Covers Next, we show how a large Steensgaard partition can be broken up into yet smaller sets that form a disjunctive alias cover for the Steensgaard partition. Unlike Steensgaard's analysis, the points-to sets generated by Andersen's analysis are not equivalence classes. An example is shown in figure 2. Here the node representing the set $\{q\}$ has out-degree three whereas in the Steensgaard points-to graph for the same example, each node has out-degree at most one. An Andersen points-to set is defined to be a set of pointers pointing to the same object in the Andersen points-to graph. Since a pointer can appear in more than one Andersen points-to sets they do not form equivalence classes. Thus we refer to them as Andersen clusters instead of partitions. However, the next result shows that they do form a Disjunctive Alias Cover.

Theorem 7 *Let pointer p belong to the Andersen points-to sets $A_1, ..., A_m$. Then $Alias(p, St_P) = \bigcup_i Alias(p, St_{A_i})$, where P is the set of pointers of the given program.*

One potential drawback of Andersen clustering is that since the clusters are not disjoint they can, in some cases, have considerable overlap with each other. Thus a single Steensgaard partition can produce a large number of Andersen clusters forming a cover. The practical implication is that although the maximum time taken to process each cluster decreases, the total time taken to process all clusters may actually increase. A solution to this problem is to identify an *Andersen Threshold* such that Andersen clustering is performed only on Steensgaard partitions larger in cardinality than this threshold. This threshold can be determined empirically. For our benchmark suite it turned out to be 60.

```
int **x, **u, **u,
**w, **z;
int *a, *b, *c;
main(){
   1a: x = &c;
   2a: w = u;
   3a: foo();
   4a: z = x;
   5a: *z = b;
   6a: bar();
}

foo(){
   1b: *x = d;
   2b: a = b;
   3b: x = w;
}

bar(){
   1c: *x = d;
   2c: a = b;
}
```

Figure 5. An Example Program

3. Scalable Context Sensitive Alias Analysis

Using Steensgaard partitioning and Andersen clustering, once the pointer aliasing problem has been reduced from the set of all pointers in the program to a small subset, we can effectively employ procedure summarization for scalable flow and context-sensitive pointer alias analysis. Indeed, since most Andersen clusters are small, the density of access for pointers belonging to a given cluster is typically low. An important implication is that summaries for a given cluster are usually small in size or even empty for most functions and can therefore be computed efficiently. We emphasize that it is clustering that allows us to leverage locality of reference. Indeed, without the clustering-induced decomposition, we would have to compute summaries for each function with a pointer access, viz., practically every function in the given program. Additionally, for each function we would need to compute the summary for all pointers modified in that function, not merely the pointers belonging to the cluster being currently processed, which could greatly affect the scalability of the analysis. Thus our technique exploits the synergy that results by combining divide and conquer with summarization.

Procedure Summaries for Context-Sensitive May-Alias Analysis. We propose a new summarization-based technique for context-sensitive pointer alias analysis. Given a context, viz., a sequence of function calls, $con = f_1...f_n$, by theorem 5 we have that pointers p and q are aliased at location l in f_n iff there exists a sequence λ of successive control locations in con starting at the entry location l_0 of the given program and leading to l such that there exists a pointer a with maximally complete update sequences from a to p and from a to q leading from l_0 to l along λ. Thus in order to compute flow and context-sensitive pointer aliases it suffices to compute functions summaries that allow us to construct maximally complete update sequences on demand. The key idea is for the summary of a function f to encode local maximally complete update sequences in f starting from the entry location of f. Then the maximally complete update sequences in context $con = f_1...f_n$ can be constructed by splicing together the local maximally complete update sequences for functions $f_1, ..., f_n$ in the order of occurrence.

We motivate our notion of summaries with the help of an example. Consider the program $Prog$ shown in figure 5. The Steensgaard partitions of $Prog$ are $P_1 = \{x, u, w, z\}$ and $P_2 = \{a, b, c, d\}$. In this case, the Steensgaard points-to graph for $Prog$ has two nodes n_1 and n_2 corresponding to P_1 and P_2, respectively, with n_1 pointing to n_2.

Consider the Steensgaard partition P_1. Note that none of the statements of function *bar* can modify aliases of pointers in P_1. This can be determined by checking that no statement of St_{P_1}

(computed via algorithm 1) occurs in *bar*. Thus for partition P_1, summaries need to be computed only for functions *main* and *foo*. Consider function foo. The effect of executing foo on pointers in P_1 is to assign w to x. Thus the local maximally complete update sequence for x leading from the entry location $1b$ of foo to $3b$ is $x = w$ which is represented via the summary tuple $(x, 3b, w, true)$. The last entry in the tuple encodes points-to constraints that are explained later on. Note that with respect to each of the locations $1b$ and $2b$, the summaries of foo are empty as the aliases of none of the pointers in P_1 can be modified by executing foo up to and including location $2b$.

Now suppose that we want the maximally complete update sequences for z leading from the entry location $1a$ of $main$ to its exit location $6a$. Since bar does not modify aliases of any pointer in P_1, the first statement encountered in traversing $main$ backwards from its exit location that could affect aliases of z is $4a$. Since z is being assigned the value of x, we now start tracking x backwards instead of z. As we keep traversing backwards, we encounter a call to foo which has the already computed summary tuple $(x, 3b, w, true)$ for its exit location $3b$. Since we are currently tracking the pointer x and since we know from the summary tuple that x takes it value from w, the effect of executing foo can be captured by replacing x with w in our backward traversal and jumping directly from the return site $3a$ of *foo* in *main* to its call site $2a$. Traversing further backwards from $2a$, we encounter $w = u$ at location $2a$ causing us to replace w with u. Since no more transitions modifying pointers of P_1 are encountered in the backward traversal, we see that $w = u, [x = w], z = x$ is a maximally complete update sequence and so $(z, 6a, u, true)$ is logged as a summary tuple for *main*. Here, $x = w$ is shown in square brackets to indicate a summary pair.

Let us now consider the set of pointers P_2. Suppose that we are interested in tracking the maximally complete update sequences for a leading from $1c$ to $2c$ in bar. Tracking backwards we immediately encounter $2c$ causing a to be replaced with b. However, when we encounter statement $*x = d$ at location $1c$, in order to propagate the complete update sequence further backwards, we need to know whether x points to b or not at $1c$. If it does then we propagate d backwards else we need to propagate b. Note that what x points to cannot, in general, be determined for function bar in isolation as it might depend on the context in which bar is called. We therefore generate the two tuples $t_1 = (a, 2c, d, 1c : x \rightarrow b)$ and $t_2 = (a, 2c, b, 1c : x \not\rightarrow b)$ accordingly as x points to b or not at $1c$, with the last entries in the tuples encoding the points-to constraints.

Definition 8 (Summary). *The summary for function f is the set of tuples $(p, loc, q, c_1 \wedge ... \wedge c_k)$ such that there is maximal complete update sequence from q to p starting at the entry location of f and leading to location loc of f under the points-to constraints imposed by $c_1, ..., c_k$. Each constraint c_i is of one of the following forms (i) $l : r \rightarrow s$ (r points-to s at l) (ii) $l : r \not\rightarrow s$ (r does not point to s at l), (iii) $l : r \sim s$ (r and s point-to the same object at l), or (iv) $l : r \not\sim s$ (r and s do not point-to the same object at l) respectively.*

Top-Down Processing. Recall from our discussion in the previous section that when computing summaries for a given Steensgaard partition P, it suffices to restrict our analysis only to pointers in V_P and statements in St_P as computed by algorithm 1. As seen above, in processing a statement of the form $*x = y$ at program location l, we need to know before hand what x points to at l. One option is to consider all possible pointers x may point to in the Andersen points-to graph. Since Andersen's analysis in both flow and context-insensitive, this may result in too many aliases which in turn may generate too many summary tuples. Because of the flow insensitivity of Andersen's analysis most of these summary tuples will likely be spurious. Thus we need to determine the set of pointers x may point to at location l accurately enough to keep the number of summary tuples small.

Towards that end, we observe that for summarization we need to consider all possible aliases of x for each path leading to l irrespective of the context. Thus when computing summary tuples, we cannot compute the objects that x may point to at location l in a context-sensitive fashion. To keep the number of summary tuples generated small, we, therefore, consider flow-sensitive and context-insensitive (FSCI) points-to sets of x. We show that the computation of these FSCI points-to sets can, in fact, be folded seamlessly into the summary computation for a given Andersen cluster P. The key observation that makes this possible is that if $*x = y$ is a statement of $Prog$ such that $x > y$, then in computing the summary tuples for P, if we encounter $*x = y$, the points-to sets for x need already have been computed beforehand. A consequence is that the summary computation for pointers in V_P needs to be carried out in a *top-down* manner in increasing order of Steensgaard depth. If, on the other hand, due to cycles in the points-to relation $*x, x$ and y, occur in the same Steensgaard partition, then we have to track points-to constraints as defined before.

Dovetailing. For top-down summary computation, we first identify the Steensgaard partitions $V_{d_1 1}, ..., V_{d_k n_k}$ of V_P, where $V_{d_j i}$ is the ith partition of V_P occurring at depth d_j in the Steensgaard points-to hierarchy, with $0 = d_1 < ... < d_k$. Next, we start computing summaries for these partitions of V_P in non-decreasing order of Steensgaard depth. In the sequel, depth refers to Steensgaard depth as defined in sec. 2.1. The key idea is to dovetail the computation of the summary tuples with the computation of the FSCI aliases as is formalized in algorithm 2. We start by computing summary tuples

Algorithm 2 Dovetailing Summary Computation with Computation of FSCI-aliases

```
1:  Input: Andersen Cluster P, pointers V_P (as computed by alg. 1) and a Steensgaard partition V_{d_1 1}, ..., V_{d_1 n_1}, ..., V_{d_k n_k} of V_P as defined above.
2:  Compute the function summaries for partitions V_{d_1 1}, ..., V_{d_1 n_1}.
3:  for i = 1 to k − 1 do
4:      for each j ∈ [1..n_i] do
5:          Compute FSCI points-to sets for pointers in V_{d_i j}
6:      end for
7:      for each l ∈ [1..n_{i+1}] do
8:          Compute summaries for V_{d_{i+1} l}
9:      end for
10: end for
```

for pointers of V_P that have the least Steensgaard depth (line 2). Next for these pointers we compute the FSCI points-to sets (lines 4-5) which, as observed before, can then be used in the computation of summary tuples for partitions of V_P one depth lower in the Steensgaard points-to hierarchy (lines 7-8). The procedure iterates over the depth of the points-to hierarchy. Note that if in computing the summaries for a partition V_{ji} of V_P at depth j, we encounter a statement of the form $*x = y$, where x points to some pointer in V_{ji}, then if $x > y$ we would already have computed the FSCI points-to set of x which would then allow us to decide how to propagate the complete update sequence backwards, else we need to track points-to constraints as formulated in definition 8. Thus the two key steps that we need are (i) the computation of the FSCI points-to sets for pointers of P at depth d given summaries for all pointers of P at depth d or less and FSCI aliases for pointers of P of depth less than d, and (ii) compute function summaries for pointers at depth d given FSCI points-to sets for pointers of depth less than d.

Computing Flow-Sensitive and Context-Insensitive Aliases. Let $p \in P$, where P is a given Steensgaard partition. Assuming that summary tuples have been computed for all pointers of V_P at depth $depth(p)$ or less, and FSCI aliases have been computed for all pointers of depth less than that of p, we now show how to compute FSCI-aliases of p at location l of function f. Note that from theorem 5, it follows that two pointers p and q are FSCI-aliased to each other at program location l if there exist paths in the control flow graph starting at the entry location of the given program *Prog* along which there exist maximally complete update sequences from a pointer a to pointers p and q. It follows that in order to compute the FSCI-aliases of p at location l, it suffices to first compute the set A of pointers such that for each $a \in A$ there is a maximally complete update sequence from a to p leading from the entry location of $Prog$ to l. Then the set of aliases of p is simply the set Q of all pointers q such that there is a maximally complete update sequence from a pointer in A to q leading from the entry location of $Prog$ to l. From the above discussion, it follows that we need two procedures: one to compute the set A from p and the other to compute the set Q from A.

Computation of A. Given a Steensgaard partition P at depth d and a program location l in function f, we show how to compute the set A of pointers such that there is a maximally complete update sequence from a pointer in A to a pointer in P leading from the entry location of $Prog$ to location l. We start with P and do a backward dataflow analysis wherein we track the *frontiers* of the maximal update sequences, i.e., the set of pointers F_m such that there is a maximally complete update sequence from each pointer in F_m to a pointer in P leading from location m to l. We assume that summary tuples (which capture the local maximally complete update sequences from the entry location of each function) have already been pre-computed for pointers of depth d or less. Then A can be computed simply by splicing together these local complete update sequences.

In order to accomplish that, we start with the set P and first compute the set of pointers $Propagate$ such that there is a local maximally complete update sequence from each pointer in $Propagate$ to a pointer in P starting at the entry location of f and leading to l, i.e., whether there is a tuple of the form $(p, l, q, cond) \in Sum_f$, where $p \in P$ and $cond$ is satisfiable (line 2). In order to test satisfiability of *cond*, we note that, by our construction, $cond$ is built from conjuncts of the form (i) $m : r \rightarrow s$ (r points-to s at m) (ii) $m : r \not\rightarrow s$ (r does not point to s at m), (iii) $m : r \sim s$ (r and s point-to the same object at m), or (iv) $m : r \not\sim s$ (r and s do not point-to the same object at m), where r and s have Steensgaard depth d or less. Then since summary tuples, and hence points-to sets, have already been computed for all pointers of Steensgaard depth d or less, the satisfiability of $cond$ can be checked at the time of computing the frontier.

If the aliases of a pointer $p \in P$ cannot be affected during any execution of f leading to location l, we need to retain all such pointers, forming the set $Retain$ (line 3), unchanged in the frontier. The frontier at the entry location of f is $P_f = Propagate \cup Retain$. We start the fixpoint computation by adding (f, P_f) to *Processing*, the set of tuples that are currently being processed (line 4). A tuple belonging to *Processing* is of the form (h, P_h), where h is a function and set P_h is comprised of pointers r such that there is a maximally complete update sequence from r to p leading from the entry location of h to location l of f. Next, to propagate the frontier backwards, we consider all functions g that call h. For each call site $call^i_{gh}$ of h in g we consider, as in the starting step, the set $Propagate^i_{gh}$ of pointers that have a maximally complete update sequence from the entry location of g to $call^i_{gh}$ (line 10). As before, any pointer of P_h for which there is no local maximally complete update sequence leading to $call^i_{gh}$ is simply propagated without modification (line 11). Note that for a function g if we have propagated the frontier for a pointer p backwards from the entry location of g once then it need not be done again. To ensure that, we maintain a map from each function g to the set of pointers which have been propagated backwards from the entry location of g. A pointer q is scheduled to be propagated backwards from the entry location of g only if it already hasn't been done previously (lines 13-16).

Algorithm 3 Computing Flow-Sensitive Context-Insensitive Aliases

1: **Input:** Set P of pointers at Steensgaard depth d, location l of function f and the entry function $entry\_function$ of $Prog$.
2: $Propagate = \{q | p \in P$ *and for some satisfiable condition cond,* $(p, l, q, cond) \in Sum_f\}$
3: $Retain = \{p | p \in P$ *and there does not exist a tuple of the form* $(p, l, q, cond) \in Sum_f\}$
4: Initialize $Processing = \{(f, Propagate \cup Retain)\}$
5: $Processed[f] = Propagate \cup Retain$
6: **while** $Processing \neq \emptyset$ **do**
7: remove a tuple $tup = (h, Q)$ from Processing
8: **for** each function g calling h **do**
9: **for** each call site $call^i_{gh}$ of h in g **do**
10: $Propagate^i_{gh} = \{q | p \in Q$, *for some satisfiable condition* $cond$, $(p, call^i_{gh}, q, cond) \in Sum_g\}$
11: $Retain^i_{gh} = \{p | p \in Q$ *and there does not exist a tuple of the form* $(p, call^i_{gh}, q', cond) \in Sum_g\}$
12: *Not-Processed* $= (Propagate^i_{gh} \cup Retain^i_{gh}) \setminus Processed[g]$
13: **if** *Not-Processed* $\neq \emptyset$ **then**
14: Insert tuple (g, *Not-Processed*) in $Processing$
15: $Processed[g] = Processed[g] \cup$ *Not-Processed*
16: **end if**
17: **end for**
18: **end for**
19: **end while**
20: **return** $Processed[entry\_function]$

Computation of Q The second step, i.e., the computation of Q from A is similar to algorithm 3 the only difference being that we now start at the entry location of the entry function of the given program and perform the dataflow analysis forward to propagate maximally complete update sequences from pointers in A till location l is reached as opposed to algorithm 3 which was carried out a similar analysis backwards.

Computing Summary Tuples. The final step is to show how to compute summary tuples for a set of pointers P in the same Steensgaard partition given the FSCI points-to sets of pointers higher in the Steensgaard hierarchy than those in P.

We analyze strongly connected components of the call graph of the given program in reverse topological order. For computing summaries for functions in a given strongly connected component Scc, we start with an arbitrary function $func$ belonging to Scc and then analyze each function in *Scc* till a fixpoint is reached. Given a pointer ptr and location loc in function $func$, we perform a backward traversal on the control flow graph (CFG) of the given program starting at loc and track maximally complete update sequences as tuples of the form $tup = (p, f, l, m, q, cond)$. Tuple $(p, f, l, m, q, cond)$ indicates that during our backward traversal we are currently at program location m and there is a maximally complete update sequence from q to p starting at m and leading to

location l in function f under the points-to constraints encoded in $cond$.

The algorithm maintains a set W of tuples that are yet to be processed and a set $Processed$ of tuples already processed. Initially, W contains the tuple $(ptr, func, loc, loc, ptr, true)$ (line 2 in alg. 5). The tuples in W are processed one by one till there are none left to process. In each processing step, we delete a tuple $(p, f, l, m, q, cond)$ from W (line 4 in alg. 5). In processing $(p, f, l, m, q, cond)$, with respect to the program statement $st : r = t$ at location m, we need to generate the new pointer value $newPtr$ that is propagated backwards as well the new condition $newCond$ under which it is propagated (line 6 in alg. 5). This is accomplished via alg. 4.

Algorithm 4 Processing a Tuple with respect to a statement

1: **Input:** Tuple $(p, f, l, m, q, cond)$, a statement $st : r = t$ at control location m and an Andersen cluster P.
2: **if** the statement at location m is of the form $st : r = t$ where $st \in St_P$ and r is at the same Steensgaard depth as q **then**
3: **if** r is a pointer variable **then**
4: **if** q is a pointer variable or q is of the form $\&s$ **then**
5: **if** $q = r$ **then**
6: $newPtr = t$ and $newCond = cond$
7: **else**
8: $newPtr = q$ and $newCond = cond$
9: **end if**
10: **else if** q is of the form $*s$ **then**
11: Compute the FSCI points-to set PT_s^m of s at location m
12: **if** $r \notin PT_s^m$ **then**
13: $newPtr = q$ and $newCond = cond$
14: **else if** s points to r at location m **then**
15: $newPtr = t$ and $newCond = cond \wedge m : s \rightarrow r$
16: **else**
17: $newPtr = q$ and $newCond = cond \wedge m : s \not\rightarrow r$
18: **end if**
19: **end if**
20: **else if** r is of the form $*u$ **then**
21: Compute the FSCI aliases PT_u^m of u at location m
22: **if** q is a pointer variable or q is of the form $\&s$ **then**
23: **if** u points to q at m **then**
24: $newPtr = t$ and $newCond = cond \wedge m : u \rightarrow q$
25: **else**
26: $newPtr = q$ and $newCond = cond \wedge m : u \not\rightarrow q$
27: **end if**
28: **else if** q is of the form $*s$ **then**
29: **if** s cannot be FSCI aliased to u at m **then**
30: $newPtr = q$ and $newCond = cond$
31: **else if** s and u can be aliased to each other at m **then**
32: $newPtr = t$ and $newCond = cond \wedge m : s \sim u$
33: **else**
34: $newPtr = q$ and $newCond = cond \wedge m : s \not\sim u$
35: **end if**
36: **end if**
37: **end if**
38: **else**
39: $newCond = cond$ and $newPtr = q$
40: **end if**

We consider two cases. First assume that the lhs expression r in $st : r = t$, is simply a (pointer) variable. By remark 1, in section 2, there are three sub-cases to consider (i) q is a pointer variable, (ii) q is of the form $*s$, and (iii) q is of the form $\&s$. In cases (i) and (iii) we know precisely during our backward traversal what q is. However in case (ii) the value of q depends on what s points to. Thus in cases (i) and (iii), if q is the same pointer as r then $newPtr = t$ (line 6 in alg. 4) else $newPtr = q$ (line 8 in alg. 4), with $newCond = cond$ in either case. If q is of the form $*s$ then using the FSCI points-to analysis presented before, we determine the set of pointers PT_s^m that s can point to at location m. Then if $r \in PT_s^m$ there are two possible scenarios: $newPtr = t$ and $newCond = cond \wedge m : s \rightarrow r$ (line 15 in alg. 4) or $newPtr = q$ and $newCond = cond \wedge m : s \not\rightarrow r$ (line 17 in alg. 4) accordingly as s points to r at m or not, respectively. If, on the other hand, $r \notin PT_s^m$, then we know that q can never equal r and so it is propagated without change, i.e., $newPtr = q$ and $newCond = cond$ (line 13 in alg. 4).

Now consider the case that r is of the form $*u$. Again we consider the three sub-cases listed above. In cases (i) and (iii), $newPtr = t$ and $newCond = cond \wedge m : u \rightarrow q$ (line 24 in alg. 4) or $newPtr = q$ and $newCond = cond \wedge m : u \not\rightarrow q$ (line 26 in alg. 4) accordingly as u points to q or not at m, respectively. In case (ii), $newPtr = t$ and $newCond = cond \wedge m : s \sim u$ (line 32 in alg. 4) or $newPtr = q$ and $newCond = cond \wedge m : s \not\sim u$ (line 34 in alg. 4) accordingly as s and u point to the same object or not, respectively.

The next step is to propagate the dataflow facts backwards to the predecessor locations of m. There are two cases to consider. First, assume that m is a return site of a function g that was called from within f. Then we have to propagate the effect of executing g backwards for $newPtr$. Towards that end, for each summary tuple of the form $(newPtr, g, exit_g, w, cond')$ we add the new tuple $(p, f, l, call_{fg}^m, w, newCond \wedge cond')$ (line 12 in alg. 5), where $call_{fg}^m$ is the call site of g corresponding to the call return at location m, to W. If there exists no such tuple and if $newPtr$ is modified semantically inside g (which can be determined by checking whether the left hand side of an assignment is FSCI-aliased to $newPtr$) then we need to propagate $newPtr$ through g and so we add the tuple $(p, f, l, exit_g, newPtr, newCond)$ to W (line 15 in alg. 5). At the same time, we also add $(newPtr, g, exit_g, exit_g, newPtr, true)$ to W (line 15 in alg. 5) so that we do not have to re-compute the above tuple the next time around we need to propagate this information back through g. Finally, if such a tuple does not exist in Sum_g nor can $newPtr$ be semantically modified inside g then executing g has no effect on $newPtr$ and so we can jump straight to the call site $call_{fg}^m$ of g matching the return site m of g. Accordingly, we add the tuple $(p, f, l, call_{fg}^m, newPtr, newCond)$ to W (line 17 in alg. 5).

For the second case, we assume that, m is not a function call return site. We consider the set $Pred$ of all the predecessor locations of m in f (line 21 in alg. 5). For each $pred \in Pred$, we form the tuple $tup = (p, f, l, pred, newPtr, newCond)$. If tup has already been processed, no action is required. else we add tup to W.

Note that algorithm 5 is interprocedural and can handle recursion. Indeed, when building the summary for a function $func$ belong to a strongly connected component Scc if we encounter a call to function $func'$ for which the necessary summary tuples haven't been computed then we first analyze that function to compute the required summary tuples via step 15. In this way we explore that part of Scc which is relevant to computing the summary tuples for $func$. By repeating the procedure for each function in Scc, we end up building summaries for each function in Scc.

Comparison with Existing Summarization Approaches. Due to top-down processing, the points-to relations for pointers which are higher in the Steensgaard hierarchy than P can be resolved at the time of building the summary tuples for P and so we do not need to consider the many possible points-to relations involving these pointers. A standard summarization approach would be *monolithic* in nature as it would track how a function could modify the points-

Example	KLOC	# pointers	Partitioning	Clustering	Time (secs)	Steensgaard Partitioning			Andersen Clustering		
						#cluster	Max	Time	#cluster	Max	Time
sock	0.9	1089	0.02	0.04	0.11	517	9	0.03	539	6	0.01
hugetlb	1.2	3607	0.3	0.5	8	1091	45	0.7	1290	11	0.78
ctrace	1.4	377	0.01	0.03	0.07	47	36	0.03	193	6	0.03
autofs	8.3	3258	0.6	1	6.48	589	125	0.52	907	27	0.92
plip	14	3257	0.7	1.2	6.51	568	26	0.57	761	14	0.62
ptrace	15	9075	0.9	1.1	16	924	96	1.46	5941	18	0.67
raid	17	814	0.01	0.06	0.12	100	129	0.03	192	26	0.03
jfs_dmap	17	14339	2.9	4.7	510	4190	39	3.62	9214	11	1.34
tty_io	18	2675	0.9	2.1	22	828	8	0.52	882	6	0.45
ipoib_multicast	26	2888	0.9	1.2	54.7	1167	15	1	1378	9	0.5
wavelan_ko	20	3117	0.6	1.4	17.68	591	44	1.2	744	19	1
pico	22	1903	2	10	$\geq 15min$	484	171	4.98	871	102	4.46
synclink	24	16355	12	18	$\geq 15min$	1237	95	26.85	3503	93	26
icecast-2.3.1	49	7490	2	12	459	964	114	15	2553	52	15
freshclam	54	1991	0.3	0.9	$\geq 15min$	157	77	0.6	740	45	0.44
mt-daapd	92	4008	1.4	6.8	$\geq 15min$	635	89	4.8	1118	83	12.79
sigtool-0.88	95	5881	2	10	$\geq 15min$	552	151	8	981	147	7
clamd	101	16639	13	34	61	1274	346	49	3915	187	41
sendmail	115	65134	125	675	$76min$	21088	596	1878	24580	193	1389
httpd	128	16180	40	89	$\geq 15min$	1779	199	35	3893	152	32

Table 1. Comparing Flow and Context-Sensitive Alias analysis without Clustering and with Steensgaard and Andersen Clustering

to relation of all pointers, i.e., those in P *and* the ones occurring higher in the hierarchy. Combinatorially, this would result in a lot more possible points-to configurations and thus larger summaries. This explain why our summarization approach is more succinct. Moreover, within each function f, we do a backward propagation and can extract precisely those points-to relations among parameters that can affect aliases of pointers in P due to execution of f. Existing approaches carry out a forward propagation and therefore consider all possible relations between parameters many of which may be irrelevant.

Computing Flow and Context-Sensitive Aliases. Finally, to compute the flow and context-sensitive aliases of pointer in a given Andersen cluster P at a given location loc in a given context $con = f_1, ..., f_k$, we follow a procedure very similar to the computation of FSCI aliases formulated in algorithm 3. The only difference is that the computations of the sets A and Q are now done with respect to a given context instead of taking the union over all contexts leading to a given program location.

Path-Sensitivity. In our analysis, we have so far ignored conditional statements rendering it path-insensitive. However, we can easily track the conditional statements encountered while building summaries as boolean expressions over program variables in the same way as we tracked points-to constraints. Thus in this case, a summary tuple would be of the form $(p, loc, q, c_1 \wedge ... \wedge c_k, con_b)$, where the additional entry con_b is a boolean expression capturing the branching constraints along an update sequence from q to p. One may chose to track the branching constraints only locally within a function as was done in [8] or globally across functions. Furthermore, BDDs can be used to represent the boolean expression con_b in a canonical fashion so as to weed put infeasible paths and hence bogus summary tuples.

4. Experimental Results

Our experiments were conducted on a variety of commonly used programs on a machine with an Intel Pentium4 3.20GHz processor and 2GB RAM. Table 1 show data for flow and context sensitive (FSCS) pointer alias analysis. The times taken for the initial Steensgaard partitioning and the bootstrapped Andersen clustering are given in columns 4 and 5, respectively. Columns 6 shows the data for FSCS-analysis without use of any clustering. Columns 8 and 9 show data when carrying out (bootstrapping) the FSCS-analysis on Steensgaard partitions, whereas columns 11 and 12 show data for FSCS-analysis on Andersen clusters which are, in turn, gotten from Steensgaard partitions. To simulate parallelization, we distribute the clusters into 5 parts (to simulate 5 machines). This is done using a greedy heuristic. First we divide the total number of pointers in the given program by 5 which gives us a rough estimate, denoted by $size_5$, of the number of pointers in each part. Then we process the clusters one-by-one and as soon as the sum of the number of pointers in each clusters exceeds $size_5$, we combine all clusters processed so far into a single part at which point we re-start the processing. The time spent on each part is the sum of the times taken to analyze all clusters individually in that part. We report the maximum time taken over all parts. The purpose of these experiments was to show the effect of bootstrapping on FSCS-alias analysis. This is in line with the goals of the paper which is to show how bootstrapping via clustering can be used to enhance existing pointer analyses. Thus our contribution is orthogonal to improving any existing pointer analysis or proposing new ones. For instance, any new pointer analysis that enhances Andersen's analysis (see for instance [9]) can be plugged directly into our framework to replace the existing Andersen's analysis. Another option is to cascade another analysis like the One-Flow analysis [3, 4] between Steensgaard and Andersen.

The original motivation for this work was static data race detection for Linux device drivers. We present data for ten drivers: $sock$, $hugetlb$, $ctrace$, $autofs$, $plip$, $ptrace$, $raid$, jfs_dmap, tty_io, and $ipoib\_multicast$. Additionally, we consider the mail transfer agents *sendmail* and *pico* (part of *Pine*). Other examples were taken from `gnu.org`. It can be seen from the results (cols. 4 vs. 7; cols. 4 vs. 10) that bootstrapping clearly enhances our FSCS-analysis. The time indicated is in seconds. While there was a clear reduction in the time taken when using clustering, the comparison between Steensgaard and Andersen clustering (cols. 7 vs. 10) is more interesting. Whereas for the *sendmail* example, the time taken decreases substantially, for the *mt-daap* example the time taken when using Andersen clustering becomes almost threefold. A closer look

Algorithm 5 Interprocedural May-Alias Summary Computation for an Andersen Cluster

Input: Andersen Cluster: P, Lock Pointer: ptr, Control Location loc, Function $func$.
Initialize W to $\{(ptr, func, loc, loc, ptr, true)\}$ and $Processed$ to $\emptyset$.
repeat
  Remove a tuple $tup = (p, f, l, m, q, cond)$ from W.
  Add tup to $Processed$
  Process tup with respect to the statement at m (alg. 4) to determine $newPtr$ and $newCond$
  if m is the entry location of f **then**
    we add $(p, l, newPtr, newCond)$ to Sum_f
  else if m is the call return site of a function call for g **then**
    if the summary tuple has been computed for $newPtr$ for the exit location $exit_g$ of g **then**
      for each tuple of the form $(newPtr,\ g,\ exit_g,\ w,\ cond') \in Sum_g$ **do**
        Add the tuple $(p,\ f,\ l,\ call^m_{fg},\ w,\ cond' \wedge newCond)$, where $call^m_{fg}$ is the call-site of g in f matching the return site m, to W and $Proc$ if it doesn't belong to $Proc$
      end for
    else if $newPtr$ can be (semantically) modified by g **then**
      Add the tuples $(p,\ f,\ l,\ exit_g,\ newPtr,\ newCond)$ and $(newPtr,\ g,\ exit_g,\ exit_g,\ newPtr,\ true)$ to W and $Proc$ if they don't already belong to $Proc$
    else
      Add the tuple $(p, f, l, call^m_{fg}, newPtr, newCond)$ to W and $Proc$ if it doesn't already belong to $Proc$
    end if
  else
    for each predecessor $pred$ of m **do**
      Add $tup = (p, f, l, pred, newPtr, newCond)$ to W and $Processed$ if tup doesn't already belong to $Proc$
    end for
  end if
until W is empty

at the *mt-daap* example reveals that there is barely any reduction in the maximum cluster size, i.e., from 89 for Steensgaard to 83 for Andersen clusters. This indicates a considerable overlap among the Steensgaard partitions because of which there is little benefit due to Andersen clustering. For the sendmail example, on the other hand, the maximum cluster size drops from 596 to 193 and accordingly so does the running time from 1878 to 1389. Thus Andersen clustering is a good option for large code with high pointer access density but little overlap among Andersen clusters which can be gauged by the difference in the maximum sizes of Steensgaard and Andersen clusters. If that difference is below a threshold then one need not resort to Andersen clustering.

References

[1] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. In *PhD. Thesis, DIKU*, 1994.

[2] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI*, 2003.

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

[4] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, 2001.

[5] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI*, 1994.

[6] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-Insensitive Points-to Analysis for C. In *SAS*, 2000.

[8] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE*, 2006.

[9] B. Hardekof and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for million lines of code. In *PLDI*, 2007.

[10] N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *PLDI*, 2001.

[11] O. Lhoták and L. J. Hendren. Scaling Java Points-to Analysis Using SPARK. In *CC*, 2003.

[12] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A Schema for Interprocedural Modification Side-Effect Analysis with Pointer Aliasing. In *ACM Trans. Program. Lang. Sys.*, volume 23, pages 105–186, 2001.

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

[14] J. Whaley and M. S. Lam. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In *SAS*, 2002.

[15] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[16] J. Whaley and M. C. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *OOPSLA*, 1999.

[17] J. Whaley, C. Unkel, and M. Lam. A BDD-based deductive database for program analysis. In *http://suif.stanford.edu/bddbddb*, 2004.

[18] R. P. Wilson and M. S. Lam. Efficient Context Sensitive Pointer Analysis for C Programs. In *PLDI*, 1995.

[19] S. Zhang, B. G. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step Towards Practical Analyses. In *FSE*, 1996.

[20] J. Zhu. Symbolic pointer analysis. In *ICCAD*, pages 150–157, 2002.

[21] J. Zhu and S. Calman. Context sensitive symbolic pointer analysis. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, volume 24, pages 516–531, 2005.