Batch Alias Analysis

Jyothi Vedurada
Dept. of CSE, IIT Madras
vjyothi@cse.iitm.ac.in

V. Krishna Nandivada
Dept. of CSE, IIT Madras
nvk@iitm.ac.in

***Abstract*—Many program-analysis based tools require precise points-to/alias information only for some program variables. To meet this requirement efficiently, there have been many works on demand-driven analyses that perform only the work necessary to compute the points-to or alias information on the requested variables (queries). However, these demand-driven analyses can be very expensive when applied on large systems where the number of queries can be significant. Such a blow-up in analysis time is unacceptable in cases where scalability with real-time constraints is crucial; for example, when program analysis tools are plugged into an IDE (Integrated Development Environment). In this paper, we propose schemes to improve the scalability of demand-driven analyses without compromising on precision.**

Our work is based on novel ideas for eliminating irrelevant and redundant data-flow paths for the given queries. We introduce the idea of *batch analysis*, which can answer multiple given queries in batch mode. Batch analysis suits the environments with strict time constraints, where the queries come in batch. We present a batch alias analysis framework that can be used to speed up given demand-driven alias analysis. To show the effectiveness of this framework, we use two demand-driven alias analyses (1) the existing best performing demand-driven alias analysis tool for race-detection clients and (2) an optimized version thereof that avoids irrelevant computation. Our evaluations on a simulated data-race client, and on a recent program-understanding tool, show that batch analysis leads to significant performance gains, along with minor gains in precision.

I. Introduction

Points-to analysis is the underlying base analysis for many other static analyses, for example, the analyses designed for callgraph construction [1], bug detection [2]–[4], compiler optimizations, program understanding [5] and so on. On one hand, there have been several whole-program points-to analyses over the last two decades [6]–[9] that compute the exhaustive solution and meet the requirements of clients that need points-to information for many parts of the code. On the other hand, many demand-driven analyses [2], [10]–[16] have been proposed that compute only necessary results for a fixed set of queries and meet the requirements of clients that need points-to information only for a few parts of the code. Whole-program analyses are expensive because they are fully exhaustive and compute the points-to information for the entire input application. Many of the demand-driven analyses are fully on-demand (work on each query independently) and may not scale well to clients with a large number of queries [17], [18], because they recompute a lot of information across different queries. However, an open question (similar to the one asked by Hind and Pioli [19]) which is of interest to the community is: what is a suitable analysis for clients that fall in-between? That is, for the clients that: (1) do not require the complete whole-program information but can trigger a significant number of queries on large applications, (2) trigger queries which are not fully on-demand (i.e., all the queries can come in a batch), and (3) are performance critical, that is, should suit IDE kind of environments. There are many such clients, for example, data-race detection analyses [4], [20], analyses for identifying refactoring opportunities [5], and so on, where the points-to or alias queries are available in a batch.

“*Pointer Analyses should be designed to be appropriate in cost and precision for specific groups of client problems.*”

–Barbara Ryder [8]

The existing demand-driven analyses [2], [10]–[16], [21] answer one query at a time (fully on-demand) and none of them provide special treatment when the queries from a client come in batch. Some of these works perform ad-hoc caching (or summarization) to avoid redundant computations across different queries without exploiting the availability of queries in advance and hence they suffer from storing unnecessary information or from missing to store useful information across different queries leading to performance losses. Su et al. [15] provide a parallel algorithm to process queries from such clients independently, but still do not take full advantage of queries arriving in batches. We are not aware of any work that efficiently answers such batch alias queries.

To address this issue of efficient alias analysis where queries come in batches, in this paper, we introduce the idea of *batch analysis*. It can efficiently serve clients with strict time constraints by modeling a batch query (a group of demand-driven queries) as a single-source multiple-destination CFL-reachability problem. To present our ideas on batch analysis, we use two demand-driven alias analyses (1) the existing best performing demand-driven alias analysis tool [2], and (2) an optimized version thereof that avoids irrelevant computations with respect to the input queries. The key step in batch analysis is identifying redundant computations across queries and carefully forming batch queries such that these computations are performed only once in a batch query. We believe that the insights behind the batch analysis can be extended to other analyses such as points-to analysis, null-pointer analysis, etc.

Our Contributions.

• We design a batch alias analysis framework that can efficiently address clients where alias queries come in batch, by avoiding redundant computations within batch queries.

• We present techniques to optimize demand-driven analyses by eliminating irrelevant analysis paths that can improve the performance of the batch analysis further. We propose a novel type-information based approach to eliminate irrelevant analysis paths when solving alias queries and thus improve the performance and precision of demand-driven alias analyses.

• We present a detailed evaluation using a simulated data-race client and on a recent program-understanding tool [5]. We show that batch analysis leads to significant performance gains, along with minor precision gains, and the elimination of irrelevant search paths improves performance and precision.

II. Preliminaries

In this section, we first introduce some notations and terminology, and then we outline the CFL-reachability-based demand-driven context-sensitive alias analysis proposed by Yan et al. [2] (implementation available [23]) as it serves as our baseline; hereafter, we refer to it as BasicAnalyzer.

Alias Analysis. We use the notation $\Phi = alias?(\mathtt{a},\mathtt{b})$ to represent an alias query between two variables `a` and `b`. To illustrate, Fig. 1 shows a modified code snippet from jfreechart-1.0.14 [22] (a chart library). Classes `BarRenderer` and `LineRenderer` extend the abstract class `Renderer`. Methods `client1`, `client3` use `LineRenderer` and methods `client2`, `client4` use `BarRenderer`. The method `apply` in `ChartTheme` applies theme/style to a chart by various chart parameters like renderer. Note that `ShadowPaint` controls the shadows drawn for the bars and hence is specific to `BarRenderer`. Suppose we are interested in finding whether the read of field `ShadowPaint:enabled` at Line 25 and the write of it at Line 22 maybe involved in a data race, the alias query is $\Phi = alias?(\mathtt{lsp}_{25}, \mathtt{fsp}_{22})$.

CFL-reachability. Context-free language (CFL) reachability is an extension of graph reachability problem in which a directed graph G with edges having labels from an alphabet Σ and a context-free language L over Σ are considered. A path p from a node n to n' in G is called an *L-path* if the word formed by concatenating the labels of edges in p, say w, is a member of L. Here, n' is said to be *L-reachable* from n. This is a popular mechanism [2], [11]–[13], [15], [16], [20], [24] to model field- and context-sensitivity.

Program Representation. *Interprocedural symbolic points-to graph* (ISPG) [2], [20] forms the basis for the CFL-reachability-based alias analysis. Fig. 2 shows the ISPG for the code in Fig. 1. The ISPG is constructed in two steps: (1) an intraprocedural SPG is built locally for each method using placeholders for the objects created outside the method. (2) The constructed intraprocedural SPGs are connected to form the ISPG. An ISPG contains three kinds of nodes: (1) nodes for local variables (shown using ellipses), (2) abstract allocation nodes representing allocated objects $o \in \mathcal{O}$ (shown using rectangular white boxes), and (3) placeholder nodes called as symbolic nodes $s \in \mathcal{S}$ (shown using shaded boxes).

Edges in ISPG: (1) The `new`, copy and store statements lead to edges similar to the ones realized in standard points-to analysis. For example, for each store statement of the form $x.f = y$, a points-to edge $o_1 \xrightarrow{f} o_2$ is added if x points to o_1 and y points to $o2$, where o_1 and o_2 can be symbolic or allocation nodes; (2) for each formal parameter f, an edge $f \rightarrow s$ is added, where symbolic node s represents the object passed from outside; (3) for each field access $v.f$, an edge $v \xrightarrow{f} s$ is added, where symbolic node s is used to represent the object that $v.f$ points to. (4) for each call site of the form $v = a_0.m(a_1, ..., a_i, ...)$: (i) an *entry edge* is added from any allocation or symbolic node that a_i points to in the caller, to the symbolic node s_i that formal parameter f_i points to in m; (ii) an *exit edge* is added from any allocation or symbolic node that a return variable points to in m to the symbolic node s that was introduced as a placeholder of the return value in the caller m. (iii) an edge $v \rightarrow s$ is added, where s (same as in (ii)) is the symbolic node representing the object returned from m. For an edge with label t, $\bar{t}$ represents the reverse edge.

BasicAnalyzer. Alias query Φ to BasicAnalyzer is performed as an on-demand single-source single-target CFL-reachability problem on ISPG. Edge labels in ISPG are field names coming from field read/write operations, and entry and exit labels. A path between any two nodes $n \in (\mathcal{O} \cup \mathcal{S})$ and $n' \in (\mathcal{O} \cup \mathcal{S})$ in ISPG has a string w formed by concatenating these edge labels. The memory alias relation between n and n' can be described using CFL $memAlias$ as:

$$\begin{aligned} memAlias \ \rightarrow\ & \bar{f}\ memAlias\ f \mid memAlias\ memAlias \\ & \mid entry \mid \overline{entry} \mid exit \mid \overline{exit} \mid \epsilon \end{aligned}$$

Thus, $memAlias$ is capturing field-sensitivity by looking for matching field names (f) in w. For example, In Fig. 1, there is a $memAlias$-path between $\mathtt{lsp}_{25}$ and $\mathtt{rsp}_{13}$ with w: "$\overline{\mathtt{sp}}$ $\overline{\mathtt{entry}_{16}}$ $\overline{\mathtt{entry}_{12}}$ $\mathtt{entry}_{13}$ $\mathtt{sp}$ $\mathtt{exit}_{13}$", and hence $\mathtt{lsp}_{25}$ and $\mathtt{rsp}_{13}$ may alias. Context-sensitivity is captured using the following grammar that matches entry and exit labels.

$$C \ \rightarrow\ (_i\ C\ )_i \mid CC \mid \epsilon \qquad \begin{aligned} (_i \ &\rightarrow\ entry(i) \mid \overline{exit(i)} \\ )_i \ &\rightarrow\ exit(i) \mid \overline{entry(i)} \end{aligned}$$

Fig. 3 shows the demand-driven context-sensitive alias analysis of BasicAnalyzer. If $\Phi = alias?(a, b)$ is an alias query on the local variables a and b, the algorithm takes (n_1, n_2) as the corresponding input query, where n_1 and n_2 are the nodes that a and b point to respectively in the ISPG. We call $\phi = (n_1, n_2)$ as an *effective query* of Φ; since a and b can point to multiple allocation/symbolic nodes, an alias query can have multiple effective queries. `checkAliasing` checks for context-sensitive $memAlias$-path reachability from n_1 to n_2 in the ISPG by performing a breadth-first traversal. The procedure maintains two stacks ($fldStk$ and $cxtStk$) to support field sensitivity and context sensitivity. The function `processEdge` processes each edge by updating $fldStk$ and $cxtStk$ with the edge label when required and returns true when it finds a context mismatch ($\Rightarrow$ the corresponding path should be ignored); detailed definition of this function is elided for space. If n_2 is reached from n_1 and the $fldStk$ is empty, it implies that variables a and b are aliases. The termination check (Line 10) does not check for $cxtStk$ to be empty as a realizable path may have unbalanced prefix and suffix edges.

Caching. Xu et al. [20] use a whole program pre-analysis to cache the reachability strings associated with all the reachable nodes of each ISPG node to compute all-pairs $memAlias$-path

```
1  void client1(){ ...
2    client3(new LineRenderer());}
3  void client2(){ ...
4    BarRenderer b = new BarRenderer();
5    b.setShadow(new ShadowPaint());
6    client4(b);}
7  void client3(LineRenderer lr){
8    ChartTheme theme = ...;
9    theme.apply(lr); ...}
10 void client4(BarRenderer br){
11   ChartTheme theme = ...;
12   theme.apply(br);
13   rsp = br.getShadow();...}
14 class ChartTheme {
15   void apply(ChartRenderer cr){
16       cr.paint();} ...}//end
17 abstract class ChartRenderer {
18   void paint(){...} ...}//end
19 class BarRenderer extends ChartRenderer{
20   ShadowPaint sp;
21   void setShadow(ShadowPaint fsp){
22     fsp.enabled=true; this.sp = fsp;}
23   void getShadow(){ return this.sp; }
24   void paint(){ lsp = this.sp;
25     if(lsp.enabled){...}else{...}; }
26 ...}//end
27 class LineRenderer extends ChartRenderer{
28    void paint(){ this...} ...}//end
29 class ShadowPaint{boolean enabled; ...}//end
```

Fig. 1: Running example. Code snippet from jfreechart [22], with minor modifications.

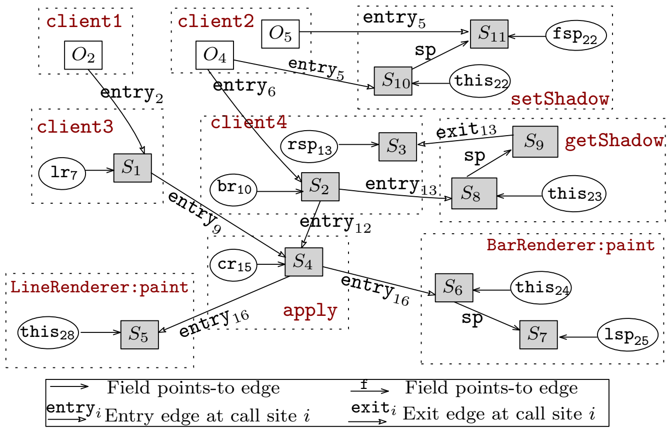


Fig. 2: Symbolic Points-to Graph for the example in Fig. 1.

```
1  function checkAliasing($n_1$,$n_2$)
2      $WL \leftarrow \{(n1, \emptyset, \emptyset)\}$;
3      while $WL \neq \emptyset$ do
4          remove $(n, fldStk_n, cxtStk_n)$ from $WL$;
5          $(fldStk, cxtStk) \leftarrow clone(fldStk_n, cxtStk_n)$;
6          foreach edge $e \in n.Edges$ do
7              $ignorePath \leftarrow$ processEdge(edge,fldStk,cxtStk);
8              if $ignorePath \neq true$ then
9                  $m \leftarrow neighbour(e, n_1)$;
10                 if $m = n_2 \wedge fldStk = \emptyset$ then
11                     if cachingOn then Results.add(($n_1, n_2$), true);
12                     return true; // $n_2$ and $n_1$ are may-aliases
13                 $WL \leftarrow \cup\{(m, fldStk, cxtStk)\}$;

14     if cachingOn then Results.add(($n_1, n_2$), false);
15     return false;
```

Fig. 3: Solving Alias Queries

```
1  if cachingOn $\wedge fldStk = \emptyset \wedge cxtStk = \emptyset$ then
2      if effective query $(m, n_2)$ is in Results then // in "cache"
3          if Results.get($(m, n_2)$) is true then return true;
4          else continue;
```

Fig. 4: Enable caching: code to be added to Fig. 3 (before Line 13).

reachability information. Yan et al. [2] in their implementation [23] of BASICANALYZER, use a caching scheme (evolved from that of Xu et al.) that caches during the demand driven analysis (in contrast to whole program pre-analysis), and caches only the results of effective queries (thus, has lower memory overhead than the work of Xu et al.) We now briefly discuss this caching scheme (termed CACHE) of Yan et al.

The code for caching gets executed upon setting an option *cachingOn*. Here, for each input query (n_1, n_2), (i) during the analysis, upon visiting a node m with empty *fldStk* and *callStk* (to respect field- and context-sensitivity), this scheme uses the prior result of the query (m, n_2) if available (shown in Fig. 4, Lines 3 and 4). (ii) in Fig. 3, before returning, the answer for (n_1, n_2) is stored (Lines 11 and 14).

III. OPPORTUNITIES FOR IMPROVEMENT

Our goal is to improve the performance of alias analysis without compromising on precision. In this section, we demonstrate two main opportunities for improving the performance of a demand-driven alias analysis, for example, BASICANALYZER. We illustrate the same by using the example program in Fig. 1. Our identified opportunities can be classified into two categories: redundant work (discussed in Section III-A) and irrelevant work (discussed in Section III-B).

A. Redundant Work

To see how BASICANALYZER performs redundant work, let us consider the example code from Fig. 1. Suppose we are interested in finding whether the read of field `ShadowPaint:enabled` at Line 25 and the write of it at Line 13 may involve in a data race and hence the alias query becomes $\Phi_1 = alias?(\texttt{lsp}_{25}, \texttt{rsp}_{13})$. Similarly, say, we are also interested to answer the alias query $\Phi_2 = alias?(\text{lsp}_{25}, \text{fsp}_{22})$. The effective query of Φ_1 is $\phi = (S_7, S_3)$ as $\texttt{lsp}_{25}$ points to S_7 and $\texttt{rsp}_{13}$ points to S_3 in the ISPG as shown in Fig. 2. Similarly, the effective query of Φ_2 is $\phi_2 = (S_7, S_{11})$. Paths π_2 and π_1 in Fig. 6 are the context-sensitive *memAlias*-paths for ϕ_1 and ϕ_2 respectively. Fig. 5 shows the two sub graphs of the ISPG in Fig. 2 containing edges of all the paths that are traversed by BASICANALYZER for answering ϕ_1 and ϕ_2 respectively. The start node during

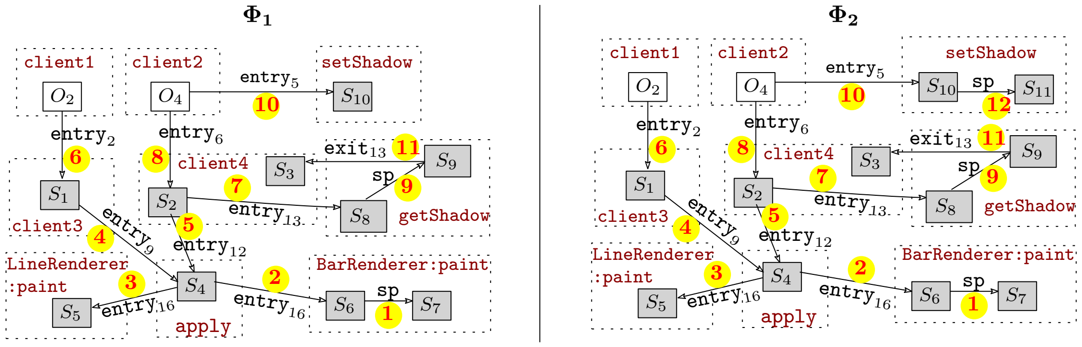


Fig. 5: Sub graphs traversed for answering $\Phi_1 = alias?(\text{lsp}_{25}, \text{rsp}_{13})$ and $\Phi_2 = alias?(\text{lsp}_{25}, \text{fsp}_{22})$.

both the traversals is S_7 and the edges are numbered according to their visiting order during the backward traversals. We can see that the only difference between the traversal of ϕ_1 and ϕ_2 is the edge 12; all other edges are redundantly traversed. Since BASICANALYZER handles each query independently, there are many redundant paths possible across different queries.

As discussed in Section I, for clients in which the input alias queries come in a batch, handling the queries independently adds a lot of overhead due to redundant traversals across queries. To address such redundant traversals, standard demand-driven analyses perform optimizations such as: caching the results of the previous queries and caching the already analyzed method summaries etc. However, as supported by our evaluation, these optimizations are not sufficient to eliminate the possible redundant traversals across different queries. If the queries are known to be available in batch, the information about the query (S_7, S_3) will be known when answering the query (S_7, S_{11}) (or vice versa). This prior knowledge can be used to avoid redundant path traversals. In this paper, we propose the idea of batch analysis framework to efficiently handle multiple queries coming in batch mode. We call our batch analysis technique as BATCHANALYZER.

B. Irrelevant Work

To see how BASICANALYZER performs irrelevant work, consider the code shown in Fig. 1. Suppose we are interested in answering only the alias query $\Phi = alias?(\texttt{lsp}_{25}, \texttt{fsp}_{22})$. Based on the points-to edges of the variables involved, using the ISPG in Fig. 2, we can see that the effective query of Φ is $\phi = (S_7, S_{11})$. To answer this query, BASICANALYZER performs backward analysis from the start node S_7 to check for a context-sensitive *memAlias*-path to S_{11}. BASICANALYZER starts with the ISPG node S_7 corresponding to $\texttt{lsp}_{25}$ and reaches S_4, the ISPG node pointed by $\texttt{cr}_{15}$ in method `apply()`, and from there reaches the nodes pointed by $\texttt{lr}_7$ and $\texttt{br}_{10}$ in methods `client3()` and `client4()` respectively; the search continues until S_{11} is found. Fig. 6 lists the different paths used by BASICANALYZER starting from S_7 until it reaches S_{11} to answer the query ϕ; the analyzer may trace these paths in any order. Clearly, we can see that π_1 is a context-sensitive *memAlias*-path from S_7 to S_{11} and is a relevant path. However, π_3 and π_4 are different.

Path π_3: This is not even a valid path and is not necessary to the query because of the following two reasons:

$$\pi_1 : \begin{tikzcd} S_7 \arrow[r, "\overline{\texttt{sp}}"] & S_6 \arrow[r, "\overline{\texttt{entry}_{16}}"] & S_4 \arrow[r, "\overline{\texttt{entry}_{12}}"] & S_2 \arrow[r, "\overline{\texttt{entry}_{6}}"] & O_4 \arrow[r, "\texttt{entry}_{5}"] & S_{10} \arrow[r, "\texttt{sp}"] & S_{11} \end{tikzcd} \checkmark$$

$$\pi_2 : \begin{tikzcd} S_7 \arrow[r, "\overline{\texttt{sp}}"] & S_6 \arrow[r, "\overline{\texttt{entry}_{16}}"] & S_4 \arrow[r, "\overline{\texttt{entry}_{12}}"] & S_2 \arrow[r, "\texttt{entry}_{13}"] & S_8 \arrow[r, "\texttt{sp}"] & S_9 \arrow[r, "\texttt{exit}_{13}"] & S_3 \end{tikzcd} \checkmark$$

$$\pi_3 : \begin{tikzcd} S_7 \arrow[r, "\overline{\texttt{sp}}"] & S_6 \arrow[r, "\overline{\texttt{entry}_{16}}"] & S_4 \arrow[r, "\overline{\texttt{entry}_{9}}"] & S_1 \arrow[r, "\overline{\texttt{entry}_{2}}"] & O_2 \end{tikzcd} ✗$$

$$\pi_4 : \begin{tikzcd} S_7 \arrow[r, "\overline{\texttt{sp}}"] & S_6 \arrow[r, "\overline{\texttt{entry}_{16}}"] & S_4 \arrow[r, "\texttt{entry}_{16}"] & S_5 \end{tikzcd} ✗$$

Fig. 6: Paths traversed on Fig. 2 for $\Phi = alias?(\texttt{lsp}_{25}, \texttt{fsp}_{22})$.

(1) Consider the subpath $S_7 \rightarrow S_6 \rightarrow S_4 \rightarrow S_1$ of π_3. The string on this subpath is $\overline{sp}\ \overline{\texttt{entry}_{16}}\ \overline{\texttt{entry}_9}$, where the type of S_1 is `LineRenderer` and the type of S_7 is `ShadowPaint`, which indicates that `LineRenderer` has a field `sp` of type `ShadowPaint`. However, from the code in Fig. 1, we can see that only `BarRenderer` has a field `sp` of type `ShadowPaint` and the subpath traversal happened because of the upcast at the virtual call at Line 16. (2) Consider the subpath $S_6 \rightarrow S_4 \rightarrow S_1$ of π_3. The string of this subpath is $\overline{\texttt{entry}_{16}}\ \overline{\texttt{entry}_9}$ and hence is a context-sensitive *memAlias*-path from S_6 to S_1. Therefore, it is expected that the types of S_6 and S_1 are compatible (same type or one is a subtype of another). However, the type of S_6 is `BarRenderer` and the type of S_1 is `LineRenderer`; both are not compatible types but are subtypes of `ChartRenderer`. Again, in this case, the invalid path is resulting due to the underlying implicit type cast.

Path π_4: Like π_3 even this path is invalid, but because of a different reason: Consider the subpath $S_6 \rightarrow S_4 \rightarrow S_5$ of π_4. It is traversing back from S_6 which belongs to method `BarRenderer:paint` to S_5 which belongs to method `LineRenderer:paint`. As the entry edges to callees `LineRenderer:paint` and `BarRenderer:paint` in the ISPG are added for the same call site at Line 16 in Fig. 1, the information from `LineRenderer:paint` cannot propagate to `BarRenderer:paint` at runtime and hence the subpath S_6 to S_5 and thereby path π_4 are invalid paths. When processing the entry and exit edges, BASICANALYZER does not recognize the paths across different target callees of the same virtual call site and hence allows such invalid paths.

To summarize, we see that even though BASICANALYZER traverses paths like π_3 and π_4, such paths are "irrelevant" for answering the query Φ. These irrelevant traversals affect the performance as well as the precision of the demand-driven

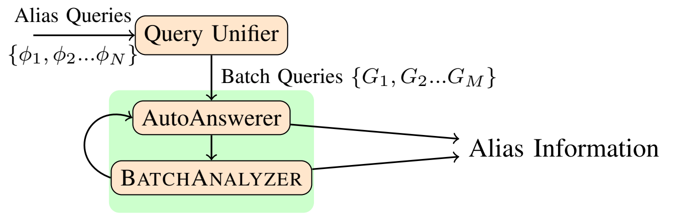


Fig. 7: Block diagram of the batch alias analysis framework.

analysis. Further, it is challenging to detect such irrelevant paths for an on-the-fly query during the analysis. Note that context-sensitivity cannot capture these invalid paths as it checks only for matching call/return pairs and all the invalid paths discussed above respects (do not violate) this check. This paper proposes a general type-information based technique to avoid processing of such irrelevant paths which can be applied to any pointer-analysis-based analysis to improve its precision and performance. We call our technique as SMARTANALYZER.

We discuss BATCHANALYZER, our batch alias analysis technique, in Section IV and SMARTANALYZER, our optimized demand-driven alias analysis technique, in Section V.

IV. BATCHANALYZER

We now discuss the details of our proposed batch analysis framework, which can use a plugged-in demand-driven analyser (for example, BASICANALYZER, or SMARTANALYZER discussed in Section V) to efficiently answer batch queries, by avoiding redundant computation. As discussed in Sections I and III, in batch analysis all the possible alias queries from a client are known in advance. Note that a naïve way to reduce the redundant computations across different queries is to cache all the intermediate traversed paths [20] and reuse them upon re-analysis. However, such techniques store a lot of information (not-scalable) which may or may not be reused in future; BATCHANALYZER overcomes these challenges.

Fig. 7 depicts our batch alias analysis framework. Our approach consists of two steps: (1) Query unification: to group the queries into a set of *batch queries* to make them more amenable to batch processing. (2) Answering batch queries by using the insights of batch analysis and the underlying demand-driven analysis engine.

Batch Query Generation. Figure 8 shows the core of our batch alias analysis framework. We group a subset G of standard alias queries to a batch query such that the redundant computations are eliminated completely across the queries in G. The function `unifyToBatchQrys` depicts how the batch queries are created. The input to the function is the set of all effective queries $\{\phi_1, \phi_2, ..., \phi_N\}$ of the input alias queries $\{\Phi_1, \Phi_2, ..., \Phi_{N'}\}$. Note that $N \geq N'$ because an alias query Φ_i can have more than one effective query if a variable in Φ_i points-to more than one symbolic/allocation node in the ISPG. Answer to Φ_i is true if at least one of its effective queries' answer is true and is false when all of its effective queries' answers are false. The function `unifyToBatchQrys` treats the effective queries (n_1, n_2) and (n_2, n_1) as the same query.

The function `unifyToBatchQrys` unifies the effective queries into groups such that all the effective queries

```
Input  : $Q = \{(n_i, m_i)\}_{i=1}^{N}$, $N$ = number of effective queries
Output: $\{r_j\}_{j=1}^{N}$ where $r_j$ stores the boolean value of $\phi(n_j, m_j)$
1  procedure batchAnalysis($Q$)
2      $S$ = unifyToBatchQrys($Q$);
3      foreach $G \in S$ processed in the order of decreasing size do
4          autoAnswer($G$);
5          batchQryCheckAliasing($G$);

6  function unifyToBatchQrys($Q$) // $Q = \{(n_i, m_i)\}_{i=1}^{K}$
7      $G \leftarrow \{\}$;
8      foreach $(n_x, m_x) \in Q$ do  {cnt($n_x$)++; cnt($m_x$)++;} ;
9      max $\leftarrow$ nodeWithMaxCnt($cnt, n_1, ..., n_K, m_1, ..., m_K$);
10     foreach $(n_x, m_x) \in Q$ do
11         if $n_x$ = max or $m_x$ = max then  $G \leftarrow G \cup \{(n_x, m_x)\}$;
12     return $\{G\} \cup$ unifyToBatchQrys($Q - G$);
```

Fig. 8: Batch Alias Analysis

```
Input  : $G = \{(n_k, m_1), (n_k, m_2), \cdots (n_k, m_M)\}$
Output: $\{r_j\}_{j=1}^{N}$ where $r_j$ stores the boolean value of $\phi(n_k, m_j)$
1  procedure batchQryCheckAliasing($G$)
2      $D \leftarrow \{m_1, m_2, ..., m_M\}$; $WL \leftarrow \{(n_k, \emptyset, \emptyset)\}$;
3      while $WL \neq \emptyset$ do
4          remove $(n, fldStk_n, cxtStk_n)$ from $WL$
5          $(fldStk, cxtStk) \leftarrow clone(fldStk_n, cxtStk_n)$
6          foreach $edge \in n.Edges$ do
7              $ignorePath \leftarrow$ processEdge($edge, fldStk, cxtStk$)
8              if $ignorePath \neq true$ then
9                  $n_2 \leftarrow neighbour(e, n_1)$;
10                 if $\exists m_j \in D$ $s.t.$ $m_j = n_2 \wedge fldStk = \emptyset$ then
11                     $D \leftarrow D - \{m_j\}$; $r_j \leftarrow true$;
12                     if $D = \emptyset$ then  return;
13                 $WL \leftarrow WL \cup \{n_2, fldStk, cxtStk)\}$;
14     foreach $m_j \in D$ do  $r_j = false$ ;
```

Fig. 9: Alias analysis for batch queries

with a common node in them are unified into the same group. Each such group becomes a batch query. However, two effective queries with a common node may still be present in two different groups. For example, if the set Q of input queries to the function are: $\{(a, b), (c, a), (d, c), (m, n), (p, q), (c, r), (a, m)\}$. Here, (c, a) can be included in the group where (a, b) is present or in the group where (d, c) is present. The function breaks such ties by greedily grouping the query with the group that can have higher number of queries in it (See lines 9-11); note: `cnt` is a local array. Thus, the set of batch queries for Q is: $\{\{(a, b), (a, c), (a, m)\}, \{(c, d), (c, r)\}, \{(m, n)\}, \{(p, q)\}\}$.

Answering Batch Queries. Given a batch query G with queries sharing a common node, and an underlying demand-driven alias analyzer (for example, SMARTANALYZER), we now discuss how BATCHANALYZER answers these queries. As shown in Fig. 8, we first check if G can be answered based on the queries answered so far; we do so by invoking the function `autoAnswer`. If not, we invoke the function `batchQryCheckAliasing`. We process the groups in the decreasing order of their size. If the group has only element, the function `batchQryCheckAliasing` invokes the `checkAliasing` function of the underlying demand-driven analysis (for example, Fig. 3) – not shown explicitly.

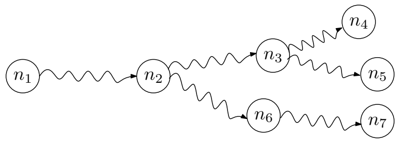


Fig. 10: Alias analysis paths in the ISPG.

In BATCHANALYZER a batch query is formulated as a single-source multiple destination CFL-reachability problem. Hence, `batchQryCheckAliasing` in Fig. 9 works similar to function `checkAliasing` in Fig. 3. The function starts from a common start node n_k and searches for the destination nodes of all individual effective queries in the batch query G. An individual query in G terminates at Line 11 when a context-sensitive $memAlias$-path is found to its destination node, otherwise it terminates at Line 14. When the function terminates, all the individual queries in G are answered. Unlike BASICANALYZER, BATCHANALYZER performs no redundant traversals across the queries in G, because BATCHANALYZER never re-traverses the common paths across the queries in G unless the retraversal is required even for BASICANALYZER to solve an individual query. Here, we can see the advantage of a batch analysis in which the redundant traversals across different queries can be avoided without caching.

`AutoAnswerer`(Fig. 7) attempts individual effective queries using prior solved queries. For example, if $alias?(n_1, n_2)$=$alias?(n_2, n_3)$=true, $\Rightarrow$ $alias?(n_1, n_3)$=true.

Thus, we can see that BATCHANALYZER avoids redundant path traversals, without caching any intermediate results, by treating a set of individual queries in a batch query as a single-source multiple destination CFL-reachability problem instead of treating it as many single-source single destination CFL-reachability problems and `AutoAnswerer` further avoids redundant path traversals.

Discussion: We briefly describe the differences/similarities between CACHE (see Section II) and the batch analysis technique, using the graph in Fig. 10 that shows ISPG nodes with $memAlias$-paths joining them. Note that we use CACHE to compare (and not its precursor [20]) as CACHE takes the input demand-driven queries into consideration for caching.

Consider the batch query $G_1 = (n_1, \{n_4, n_5, n_7\})$. Here, the batch analysis will traverse each of the paths in the graph only once. In contrast, CACHE re-traverses the paths $n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_3$ across the queries in G_1: (n_1, n_4), (n_1, n_5) and (n_1, n_7); here, path $n_2 \rightsquigarrow n_3$ may also be traversed when answering (n_1, n_7) because the traversal is performed in BFS (breadth-first search) manner. However, say, if a batch query $G_0 = (n_3, \{n_4, n_5\})$ is solved before G_1, batch analysis re-traverses the paths n_3 to n_4 and n_3 to n_5, whereas CACHE reuses those paths if the individual queries in G_0 are solved before solving the individual queries in G_1. Hence, in theory, neither of the techniques is more precise than the other (see Section VI for an empirical comparison). Note that caching can complement batching. For example, the answers to the individual queries (n_3, n_4) and (n_3, n_5) of G_0 can be cached, and reused when solving a latter batch query, like G_1.

Statement	Type Information
Alloc: `x=new()` Copy: `x=y`	$\tau_1 = typ(\texttt{x}) \wedge \tau_2 = (typ(\texttt{new()}) \vee typ(\texttt{y}))$ $\Rightarrow \tau_2 \in cmpt(\{\tau_1\}) \wedge \tau_1 \in cmpt(\{\tau_2\})$
Load: `x=y.f` Store: `y.f=x`	$\tau_1 = typ(\texttt{x}) \wedge \tau_2 = typ(\texttt{y})$ $\Rightarrow \tau_2 \circ \tau_1 [\tau_2\ encloses\ \tau_1]$
Call: `r=` $\texttt{a}_0.\texttt{m}(\texttt{a}_1,..,\texttt{a}_i,..)$	Equivalent to assignments: $\texttt{f}_i = \texttt{a}_i$, $\texttt{r} = \texttt{m}_{ret}$; where $\texttt{f}_i$ is the i^{th} formal parameter of m and $\texttt{m}_{ret}$ is the variable returned in m

Fig. 11: Type relations in the statements that affect points-to information. $typ(e)$ returns the static type of expression e.

V. SMARTANALYZER

In this section, we use a novel type-based approach to reduce the irrelevant work performed during demand-driven analysis. Our approach is based on identifying relevant types for the queries and by detecting edges that need not be traversed (termed *imaginary adjacent ISPG edges*).

A. Relevant Types

Our goal now is to provide a general scheme to avoid demand-driven alias (or points-to) analysis via invalid paths that occur due to upcasting and downcasting in codes written in languages that support subtype polymorphism.

Our strategy is based on identifying the set of relevant types for a given input query Φ. To avoid traversing irrelevant paths during backward analysis, we process a node only if the node has a relevant type with respect to Φ. Our approach is based on the following two definitions.

Definition 1 (Compatible Types). *Given a type τ, all the sub-types and supertypes of τ are the compatible types of τ. Thus, for a set of types T, we compute its set of compatible types $cmpt(T) = T \cup \{t \mid t$ is a sub type or a super type of τ, where $\tau \in T\}$.*

Definition 2 (Type Enclosing). *We say a type τ' encloses a type τ'' if there exists a sequence $\tau_1 \circ \tau_2 \circ \tau_3 \circ \ldots \tau_k$ such that $\tau_1 = \tau'$, $\tau_k = \tau''$ and for all $1 \leq i < k$, there is a field of type τ_{i+1} in τ_i (represented by $\tau_i \circ \tau_{i+1}$).*

Fig. 11 shows the compatible types and type enclosing relations in the program statements that affect the points-to information. For example, for an assignment (copy) statement, the LHS variable's type belongs to the compatible types of the RHS variable and vice versa. For a store statement, the base variable's type of the LHS field access encloses the type of the RHS variable. The types of actual arguments and of LHS variable at a call statement relate to the types of formal parameters and of a return expression in the callee, in a manner similar to that of a copy statement. As the edges in ISPG are constructed by processing these program statements (see Section II), entry/exit edges represent compatible type relations and field edges represent type enclosing relations.

Definition 3 (Declared Types). *If $\Phi = alias?(\texttt{a}, \texttt{b})$ is an input query and τ_1 is the declared-type of variable* a *and τ_2 is the declared-type of variable* b*, then we define the set of declared-types $\{\tau_1, \tau_2\}$ as $Dtype(\Phi)$.*

Definition 4 (Relevant Types). *We define the set of relevant types for a given query Φ as: cmpt($\mathcal{D}$)$\cup$cmpt($\mathcal{E}$), where $\mathcal{D}$ is the Dtype(Φ) and $\mathcal{E} = \{\tau \mid \tau$ encloses τ' and $\tau' \in$ cmpt($\mathcal{D}$)$\}$.*

Let $\Phi = alias?(a, b)$ is the input query and $\phi = (S_1, S_2)$ is one of the effective queries of Φ. A context-sensitive $memAlias$-path of Φ starts from S_1 whose declared-type is say τ_1 and passes through intermediate nodes connected via field edges that represent type enclosing relations w.r.t. τ_1, or entry/exit edges that represent compatible type relations and finally reaches S_2 whose declared type is say τ_2. Thus, the set of relevant types for Φ (where, *Dtype*(Φ) $= \{\tau_1, \tau_2\}$) captures the set of types of the nodes that can appear in a context-sensitive $memAlias$-path of Φ. A context-sensitive $memAlias$-path has the following key property:

Property 1. *If a path π is a context-sensitive $memAlias$-path for a query Φ, then a node n appears in π iff static-type(n) $\in$ Rtype, where Rtype=set of relevant types for Φ.*

Our SMARTANALYZER computes relevant types and uses Property 1 to answer a query Φ. It processes the edges (by calling `processEdge()`, Section II) connected to a node in the worklist only if its type is in the set of relevant types of Φ and thus avoids the irrelevant paths that can be taken by BASICANALYZER. We now demonstrate the same on the example query $\Phi = alias?(\texttt{lsp}_{25}, \texttt{fsp}_{22})$ from Section III. Here, *Dtype*(Φ) $= \{$`ShadowPaint`$\}$, and the set of relevant types of Φ is *Rtype* $= \{$`ShadowPaint, BarRenderer, ChartRenderer, Object`$\}$. *Rtype* includes `BarRenderer` as it encloses `ShadowPaint` and includes $\{$`ChartRenderer, Object`$\}$ because they are the comaptible types of `BarRenderer`. Consider the paths π_1 and π_3 in Fig. 6 traversed by BASICANALYZER for answering the query Φ. Path π_1 is a context-sensitive $memAlias$-path and hence the types of nodes in path π_1 $\{$`ShadowPaint, BarRenderer, ChartRenderer`$\}$ are included in the set *Rtype*. As `LineRenderer`, the type of node S_1 in π_3, is not included in the set *Rtype*, path π_3 cannot be a context-sensitive $memAlias$-path for Φ and hence it is not relevant to Φ. Similarly, path π_4 in Fig. 6 is not relevant to Φ. SMARTANALYZER avoids traversing through paths π_3 and π_4, thereby reducing the search space.

Note that the idea of relevant types is not specific to CFL-reachability-based demand-driven alias analysis and is directly applicable to any demand-driven points-to/alias analysis as the data-flow in all these analyses is similar.

B. Imaginary Adjacent Edges

Our goal now is to detect and thereby avoid invalid path traversals discussed in Case 2 in Section III-B, which happens whenever there are multiple target callees at a virtual call site.

Let $\phi = (S_1, S_2)$ be an effective alias query. Let us consider a method m_{caller} which calls a method m_{callee} at a call site x. Let p be a path from S_1 which has an edge $e1 = (n_1, n_2)$ in it such that n_1 is in m_{callee} and n_2 is in m_{caller}. Path p is an invalid path if the edge e_1 has an adjacent edge $e_2 = (n_2, n_3)$ in p such that n_3 is in m'_{callee}, where $m'_{callee} \neq m_{callee}$ is another target callee of m_{caller} at the same call site x. Hence, we focus on detecting such imaginary adjacent edges to avoid invalid paths. In particular, we look for adjacent edge pair (e_1, e_2) in path p: $\overset{S_1}{\bullet} - - \overset{n_1}{\bullet}\overset{e_1}{\rule{2em}{0.4pt}}\overset{n_2}{\bullet}\overset{e_2}{\rule{2em}{0.4pt}}\overset{n_3}{\bullet} - - \overset{n_k}{\bullet}$ (edge directions are omitted for simplicity), where n_1 is in m_{callee}, n_2 is in m_{caller}, n_3 is in m'_{callee}, and m_{callee} and m'_{callee} are the target callees of m_{caller} at the same call site x. We take this into consideration by noting that as we are interested in a traversal from a callee to a caller and again to a callee, e_1 and e_2 should have opposite directions. Thus, we ignore paths with (e_1, e_2) pair having edge labels as $(\overline{entry_x}, entry_x)$ or $(exit_x, \overline{exit_x})$. To identify such adjacent edge pair combinations, SMARTANALYZER memoizes the last processed edge. It marks the edges with unique IDs to store the call site information. Thus, SMARTANALYZER further reduces the search space over that obtained using relevant-types, by eliminating invalid paths due to imaginary adjacent edges.

VI. IMPLEMENTATION AND EVALUATION

Implementation. Our algorithms are implemented [25] by extending the Soot [26] framework We consider the following seven variants for our evaluation. **BASIC:** Fig. 3, existing BASICANALYZER [2]; **RT:** Fig. 3 with use of relevant types (Section V-A); **VC**: Fig. 3 with only valid paths at virtual call sites (Section V-B); **SMART:** SMARTANALYZER, Fig. 3 with extensions in RT and VC; **BATCH:** BATCHANALYZER using BASIC in Fig. 8 and **S-BATCH:** BATCHANALYZER using SMART in Fig. 8. **CACHE:** Fig. 3 with Fig. 4 and with caching option enabled. We used the implementations of Yan et al. [2], [23] for BASIC and CACHE. Note that we do not choose the recent demand-driven works [14], [21] as baselines for our work, as they are not suitable (we found them to take significantly more time, compared to BASIC) for data-race kind of clients that require simpler alias information.

We investigate the following three research questions: **RQ1:** Is batch analysis efficient compared to a conventional demand-driven analysis? **RQ2:** Is batch analysis beneficial compared to caching? **RQ3:** What is the impact of eliminating irrelevant paths on the precision and the performance of a demand-driven analysis?

Experimental Setup. We have run all our experiments on an AMD 2.3GHz machine by allocating 64GB heap space to the JVM. For RQ1, we evaluate our techniques on two clients: (1) a data-race detection tool (simulated), and (2) a refactoring tool (existing). For RQ2 and RQ3, we only use the data-race detection client. For all the RQs, the set of all the queries for a benchmark is collected and passed as an argument to the function `batchAnalysis` (see Fig. 8).

We simulate a data race detector client as done by Yan et al. [2] by considering all the queries of the form $alias?(\texttt{x}, \texttt{y})$ where for some field `f`, `x.f` and `y.f` are accessed in two program statements (Soot's jimple IR statements) of the application code with at least one statement in the two being a field write operation. For this client, we evaluate our technique using twelve Java programs from the DaCapo benchmark

DaCapo 2006		DaCapo 2009		Used by Client	
Bench	#RM	Bench	#RM	Bench	#RM
antlr	24190	avrora	10693	avrora	33780
bloat	13365	luindex	9730	fop	48780
chart	29017	lusearch	9612	javaGeom	27166
eclipse	12107	pmd	13448	JFreeChart	29137
fop	25252			jOcular	33059
hsqldb	28231			Rackj	15789
jython	18926			sunflow	28530
xalan	24121			UR	24434

Fig. 12: Benchmarks used for the experiments. #RM (#Reachable methods) is as computed by Spark [1].

suite [27]. We have used (i) four benchmarks from the latest DaCapo version (dacapo-9.12-MR1-bach.jar) – all the ones that could be processed by the TamiFlex [28] (to eliminate reflective calls) and Soot combination, and (ii) eight non-overlapping ones from DaCapo 2006, as listed in the first and the third columns of Fig. 12. The figure also lists the number of callgraph reachable methods as computed by Spark [1].

Next, we consider an existing client Auto-SCST [5], a program understanding tool which automatically identifies refactoring opportunities for two important refactorings 'replace type code with subclass' (SC-refactoring) and 'replace type code with state' (ST-refactoring) that are required for performing 'replace conditional with polymorphism' refactoring. For many client analyses such as Auto-SCST [5], CHORD [4], and so on, the alias queries can come in batches. Auto-SCST has strict time budgets to identify the refactoring opportunities because it is an Eclipse-plugin (IDE plugin). The tool classifies an identified refactoring opportunity, represented using the class and field pair $\langle C, f \rangle$, for ST-refactoring if there is a control path from a read dereference $x.f$ to a write dereference $y.f$, both the dereferences refer to the field f of C, and x and y are aliases. Since the refactoring opportunities are independent of each other, and all the opportunities are identified before displaying to the developer, the set of all the queries is available (as input to `batchAnalysis`, Fig. 8). For Auto-SCST, we use the eight open-source applications used by Vedurada and Nandivada [5], [29]. The last two columns in Fig. 12 show the details of the applications. Note that for the overlapping benchmarks, the entries in fourth column are different from that of the ones in the last column, owing to the differences in benchmark versions and the callgraph configurations.

For all the experiments, we use the cutoff used by by Yan et al. [2] and set the budget as 75000 traversed ISPG nodes for: (i) each demand-driven query in case of BASIC and (ii) each batch-query in case of BATCH and S-BATCH. We also enable summarization (to memoize intermediate method-level summaries/results) for all the library methods.

A. RQ1: Benefits of Batch Analysis

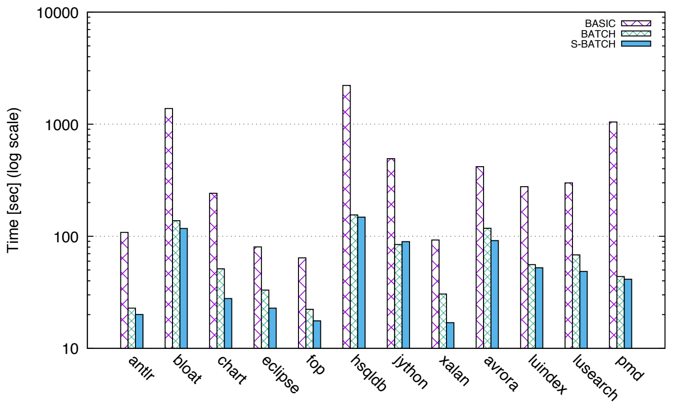
We first discuss about the analysis times of BASIC, BATCH, and S-BATCH, and then about their precision on the data-race detection client followed by on the client Auto-SCST. Fig 13 and Fig. 14 show the comparison of the time taken by the implementations and of their precisions, respectively. The second column in Fig. 14(a) lists the number of alias queries coming from the data-race client for each benchmark and the third column lists the number of their corresponding batch queries. We can see that the number of batch queries are quite less than that of the actual queries; this is because of the query-unifier step in the batch analysis framework. Similarly, in Fig. 14(b), columns 2 and 3 list the number of alias queries generated by Auto-SCST and resulting batch-queries.

Results on Data-race Detection Client. Fig. 13(a) shows the analysis times of all the three implementations to answer the alias queries. BATCH runs geomean 77.51% (up to 95.8%) faster than BASIC and S-BATCH runs geomean 83.14% (up to 96.1%) faster than BASIC. S-BATCH performs geomean 12.13% (up to 45.64%) faster than BATCH showing the impact of eliminating irrelevant path traversals.

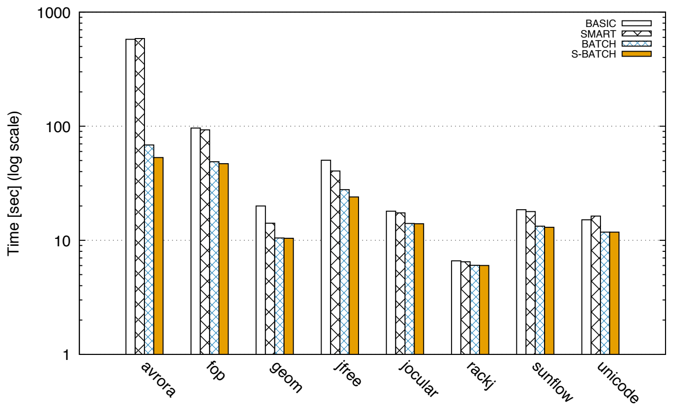
We have studied the number of budget exceptions of all the three implementations (detailed plots skipped for space). We found BATCH and S-BATCH lead to 85.52%, and 87.8% (geomean) less budget exceptions than BASIC, respectively – points to the performance improvements and scalability due to the batch alias analysis. Note that the percentage reduction in the number of budget exceptions is due to the batching of queries. For example, say, out of a given set of ten demand-driven queries six got timed-out with BASIC. If these ten are grouped as a single batch query then BATCH would timeout only once, as they are all solved together. Thus, the number of budget exceptions reduce from six to one (hence would lead to overall faster execution). Note that BASIC and BATCH would give the same answer for the remaining four.

Columns 4, 8 and 9 in Fig. 14(a) show the precision of all the three implementations. Each entry shows the number of 'true' answers of the data race queries and thus indicate the number of possible data races; lower the number, higher the precision. BATCH reports up to 1.12% less data races than BASIC (see `avrora`). The precision of BATCH and BASIC varies slightly because BATCH transforms some effective queries from (a, b) to (b, a) form during the unification step (see Section IV). Because of this swapping, BATCH may avoid hitting budget exceptions in some cases – leads to improved precision. However, such swapping may also lead to newer budget exceptions or encounter static globals in the backward analysis and conservatively answers 'true' – leads to reduced precision. For example, for the program `hsqldb`, BATCH reports two more data races than BASIC; but we found that number of such cases have been very minimal. Similar to BATCH, S-BATCH reports up to 10.22% less data races than BASIC (for `xalan`) and reports 0.28% more data races for `luindex`. We found that, overall, BATCH and SBATCH report minor improvements in precision: 0.07% and 0.61% (geomean) less data races than BASIC, respectively.

Summary. Overall, compared to BASIC, over DaCapo benchmarks, BATCH and S-BATCH run 77.5% (up to 95.8%) and 83.1% (up to 96.1%) faster, respectively, with minor precision gains; attests to the importance of batch analysis.



(a) Impact on data race queries (on DaCapo benchmark suite).



(b) Impact of batch analysis on queries from Auto-SCST.

Fig. 13: Timing comparison (in seconds).

1	2	3	4	5	6	7	8	9
Bench	#Alias Queries	#Batch Queries	Precision					
			BASIC	RT	VC	SMART	BATCH	S-BATCH
antlr	8594	283	4941	4941	4941	4941	4941	4941
bloat	51158	974	51110	51110	51103	51103	51109	51102
chart	2475	338	2407	2248	2407	2242	2407	2243
eclipse	2516	306	2418	2418	2418	2418	2414	2414
fop	899	105	899	899	899	899	899	899
hsqldb	15422	729	14730	14730	14724	14724	14732	14726
jython	9413	615	9103	9103	9103	9103	9090	9090
xalan	988	105	930	930	835	835	930	835
avrora	16720	527	15920	15920	15911	15911	15742	15741
luindex	2627	295	2531	2531	2531	2531	2538	2538
lusearch	1913	253	1884	1884	1884	1884	1883	1883
pmd	35547	365	35531	35531	35531	35531	35531	35531

(a) Precision on data race queries.

1	2	3	4	5	6	7
Bench	#Alias Queries	#Batch Queries	Precision			
			BASIC	SMART	BATCH	S-BATCH
avrora	6136	248	4984	4984	4761	4761
fop	781	155	519	518	540	539
javaGeom	158	13	137	137	134	134
JFreeChart	748	146	654	654	654	654
jOcular	243	25	180	180	185	185
RackJ	75	9	29	29	29	29
sunflow	75	22	75	75	75	75
UR	62	14	36	36	42	42

(b) Precision on queries from the client Auto-SCST.

Fig. 14: Precision comparison in terms of the number of aliases detected (lower the better).

Results on Auto-SCST. SMART, BATCH and S-BATCH show similar performance and precision improvements over BASIC on Auto-SCST. Fig. 13(b) shows the analysis times of all the four implementations on Auto-SCST. SMART, BATCH and S-BATCH run 1.75%, 31.94%, and 33.3% (geomean) faster than BASIC, respectively (and up to 29.35%, 88.15%, and 90.81% faster than BASIC, respectively). In terms of number of budget exceptions, SMART, BATCH and S-BATCH have 3.7%, 73%, and 78.03% (geomean) less budget exceptions than BASIC, respectively.

In Fig. 14(b), columns 4-7 list the number of 'true' answers for the alias queries. SMART, BATCH and S-BATCH report up to 0.19% (`fop`), up to 4.47% (`avrora`) and up to 4.47% (`avrora`), respectively, less aliases than BASIC. In `UR`, `fop`, and `jocular` the reason for precision losses of BATCH and S-BATCH is again because of the swapping of queries in the unification step. We found that, overall, SMART reports 0.02% (geomean) less data races than BASIC, and BATCH and S-BATCH report 5.46%, and 5.44% (geomean) more data races than BASIC. The geomean percentages for BATCH and S-BATCH are mainly biased by the precision differences in UR.

Summary. The performance improvements show the benefits of batch analysis over demand-driven analysis and the need for a batch analysis framework, for real-world clients like Auto-SCST, to answer alias queries in batch mode.

Discussion. For the tested benchmarks, auto-answerer (see Section IV) had no impact. But in practice, we can easily visualise cases where auto-answerer can be impactful. Hence, we keep the auto-answerer step intact, especially considering minimal overheads due to auto-answerer.

B. RQ2: Batch Analysis vs Caching

We now compare BATCH and S-BATCH against CACHE (see Section II) to show the performance of our batch analysis technique against caching. As it is known, the performance of CACHE may depend on the order of queries. For each benchmark, we use the data-race queries obtained by iterating over the methods in the order as given by the SPARK call graph (as used by Yan et al. [2]).

Fig. 15 shows the performance of CACHE against BASIC, BATCH and S-BATCH on the queries from the data-race detection client. CACHE performs 20.37% (geomean) better than BASIC whereas BATCH and S-BATCH perform 77.51% and 83.14% (geomean) better than BASIC, respectively. Clearly,

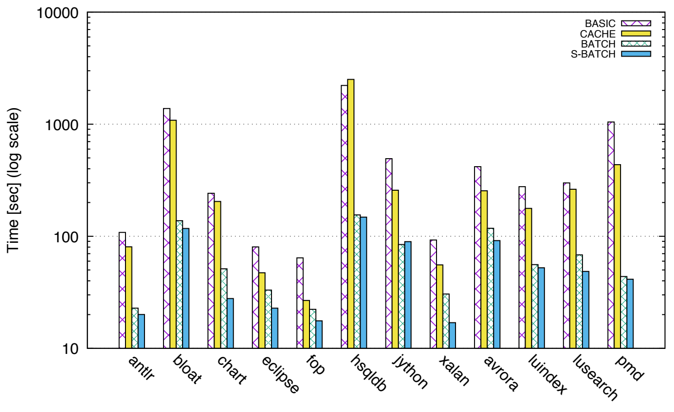


Fig. 15: Caching against batch analysis on data race queries.

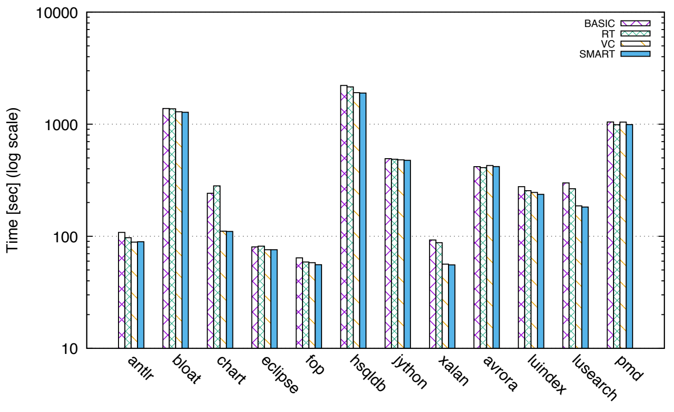


Fig. 16: Impact of irrelevant path elimination (DaCapo suite).

our batch analysis techniques outperform CACHE; In terms of the number of budget exceptions, CACHE, BATCH and S-BATCH have 48.48%, 85.52% and 87.8% (geomean) less budget exceptions than BASIC, respectively, asserting the improvements in analysis timings. In case of `hsqldb`, we see that caching shows negative effects which can happen when the benefits from reusing the cached entries get subsumed by the cost of searching the cached entries. We have also tried enabling caching for BATCH and for S-BATCH, and we observed that caching has minimal impact on batching techniques (BATCH improved by 0.38%, S-BATCH's performance decreased by 7.6% (geomean)). This shows that our batch analysis technique subsumes most of the benefits of caching. In terms of precision, CACHE reports 0.02% (geomean) more data races than BASIC, whereas BATCH and SBATCH reported 0.07% and 0.61% (geomean) less data races than BASIC, respectively. Note that the reason for precision loss for CACHE (compared to BASIC) is because of not differentiating the cached queries of form (a, b) with their swapped form (b, a) during reuse (similar to batch analysis).

Summary. Overall, compared to CACHE, BATCH and S-BATCH run 54.54% and 70.3% faster, respectively. We argue that batch analysis technique mostly subsumes the benefits of caching and outperforms it.

C. RQ3: Performance of SMARTANALYZER

We first discuss about the analysis times of the implementations BASIC, SMART, RT and VC, and then about their precision. Fig. 16 shows the analysis times of all the four implementations to answer the alias queries. SMART performs 10.97% (geomean) better than BASIC (up to 54.36% better). We explain the reasons behind the performance difference by studying the performance of RT and VC.

In Fig. 16, it can be seen that for most of the benchmarks RT and VC perform better than BASIC. The reason for RT and VC not showing performance benefits over BASIC in some cases is the following: BASIC may conservatively answer an alias query as 'true' via an irrelevant path, whereas RT or VC may try for the precise answer 'false' by searching a much longer 'relevant' search space – can be expensive. For example, in the case of the program `chart`, RT takes 16.3% more time than BASIC (also realizes improved precision, see Fig. 14(a)). In Fig. 17, we show that the overheads of the RT implementation cannot be the reason for such performance degradation. Fig. 17 lists the total time taken by RT to find the relevant types for all the queries (in last column) and the average number of relevant types for a query (in third column). We can see that the average time taken by RT to find relevant types is almost negligible. RT performs up to 11.3% (lusearch) better than BASIC; geomean = -0.43%; the negative value is mainly because of the difference in performance on `chart`. VC performs geomean 9.01% (up to 54%) better than BASIC.

Bench	Total types	Avg Relevant types	% Avg Relevant types	Time (msec)
antlr	5865	4109	70.0	722
bloat	3204	2191	68.4	948
chart	6642	4722	71.0	1240
eclipse	3261	2255	69.1	513
fop	8109	5850	72.1	646
hsqldb	6349	4478	70.5	1401
jython	4299	3127	72.7	975
xalan	6404	4543	70.9	530
avrora	3418	2326	68.0	855
luindex	3136	2066	65.8	383
lusearch	3121	2045	65.5	363
pmd	3792	2453	64.6	413

Fig. 17: Details on the computed relevant types.

In Fig. 14(a), columns 4-7 show the precision of all the four implementations. SMART reports up to 10.2% less data races than BASIC (for `xalan`). SMART can be more precise than BASIC when SMART finishes the search within the time budget and: (1) BASIC fails to finish because of irrelevant traversals, or (2) BASIC conservatively hits a static global or a destination node via an irrelevant path. For the `chart` program, the 16.3% performance loss of RT can be attributed to the 6.61% precision improvements for that program. VC also shows up to 10.2% precision improvements. We found that, overall, RT, VC and SMART report 0.18%, 0.23% and 0.47% (geomean) less data races than BASIC, respectively.

Summary. Overall, compared to BASIC, SMART runs 10.97% (up to 54.36%) faster with minor precision gains, on

DaCapo benchmarks. It shows that demand-driven analyses can be made performant by avoiding irrelevant path traversals.

VII. Related Work

There exist many prior works on alias/points-to analysis. Sridharan et al. [7] present a survey of the alias analysis techniques for object-oriented programs. Ryder [30] and Hind [8] present insightful discussions on different dimensions on pointer analysis. In this section, we mainly focus our discussion to the works that are closely related to our approach.

Demand-driven Points-to/Alias Analyses. Many researchers [11]–[13], [15], [20] have focused on improving the scalability of demand-driven points-to analysis based on CFL-reachability to suit the environments like just-in-time (JIT) compilers and interactive development environments (IDEs). Past works [2], [16] further improved the performance of CFL-reachability based solution by answering the alias queries directly without computing the points-to information and show that for answering may-alias queries, a demand-driven alias analysis is more efficient than a demand-driven points-to analysis. Boomerang [14] is a demand-driven, flow-, field- and context-sensitive pointer analysis built on top of the IFDS (Interprocedural Finite Distributive Subset problems) framework. It supports all-alias (to find all aliases of a given variable along with its points-to information) queries that are required for the clients like taint analysis. Since it requires all-alias queries, in cases of simple demand-driven points-to/alias queries, it can perform more work than that required to answer the input demand-driven queries. Recently Späth et al. [21] have presented a demand-driven, flow-, field- and context-sensitive pointer analysis which encodes context-sensitivity and field-sensitivity as a separate "synchronized" CFL reachability problems using the concept of synchronized pushdown systems (SPDS). All these techniques answer one query at a time (fully on-demand) and none of them exploit the resulting advantages when the queries come in batch. These works perform ad-hoc caching (or summarization) to avoid redundant computations across different queries without exploiting the availability of queries in advance in batch mode. Hence they suffer from storing unnecessary information and/or from missing to store useful information across different queries leading to performance losses.

We overcome these challenges by proposing a batch analysis framework that can handle multiple queries in a batch mode. We instantiate a "batch version" of the popular demand-driven analysis of Yan et al. [2]. We have observed the recent demand-driven works [14], [21] although more precise than that of Yan et al. [2], do not suit for clients where the number of alias queries is large. Further, most of the techniques discussed above suffer from the issue of irrelevant path traversals, which is addressed in this paper.

Analyses that Handle Queries in Batch. Su et al. [15] proposed a parallel algorithm to CFL-reachability-based demand-driven, context-sensitive points-to analysis to meet the requirements of clients where the queries come in batch. The technique processes the queries independently on different cores allowing some data-sharing across different queries based on few heuristics. In this paper, we propose a sequential algorithm which processes multiple queries together in one pass as a batch query. Hence, our technique does not require data sharing across individual queries in a batch query.

There have been prior works (not demand-driven) that present target client-specific solutions [31], improving the precision based on a complete set of queries [32]–[35]. In contrast, we present a scheme to improve the performance and precision by exploiting the underlying features of batch queries. Their techniques are complimentary to our approach.

Type-based Optimizations in Points-to/Alias Analyses. Diwan et al. [36] use type information and propose different alias analyses for Modula-3, a statically-typed type-safe language. For Java, many past works [1], [7], [11] perform type-based checks to avoid invalid data-flow paths and thus improve the precision and efficiency of the points-to analysis. The work of Sridharan et al. [11] (hereafter referred to as SRI) uses type-based optimizations that subsumes the ones used by Lhoták and Hendren [1], and Sridharan et al. [7]. The type-based checks of SRI use compatible types to avoid marking possible aliasing nodes in two phases: (1) during a whole-program pre-analysis by avoiding the addition of some "match" edges, ahead of time (not dependent on demand-driven queries), (2) on-the-fly when answering a points-to query by not considering nodes whose types are incompatible with the type of the query variable. In contrast, we consider enclosing types along with compatible types (used in SRI) for constructing the set of relevant types with respect to the given demand-driven queries and use them to avoid traversing irrelevant paths in the ISPG.

VIII. Conclusion

Demand-driven analyses can be costly when applied on large systems where the number of queries is high. This is not acceptable in cases when the clients of the analyses are plugged into an IDE-kind of resource-bound environments.

We propose the idea of batch alias analysis that can efficiently address such clients by avoiding redundant computations within batch queries. Batch analysis suits the environments with strict time constraints, where the queries come in batch. Our proposed techniques (publicly available [25]) improve the performance by avoiding redundant and irrelevant computation. Our evaluation shows that the proposed batch analysis leads to significant performance gains, along with minor gains in precision. We also show that the elimination of irrelevant search paths can improve the performance and the precision of the demand-driven alias analysis of Yan at al. [2] (the best one suitable for race-detection type of clients).

Acknowledgment

We thank the shepherd, the PC chairs and all the reviewers for their valuable comments. We thank Dacang Yan, Harry Xu and Atanas Rountev for making their code publicly available [23]. Jyothi Vedurada is supported by the TCS Research Scholarship Program for her doctoral studies.

REFERENCES

[1] O. Lhoták and L. Hendren, "Scaling Java Points-to Analysis Using Spark," in *Compiler Construction*, G. Hedin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 153–169.

[2] D. Yan, G. Xu, and A. Rountev, "Demand-driven Context-sensitive Alias Analysis for Java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 155–165. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001440

[3] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 133–143.

[4] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 308–319. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134018

[5] J. Vedurada and V. K. Nandivada, "Identifying Refactoring Opportunities for Replacing Type Code with Subclass and State," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 138:1–138:28, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276508

[6] L. O. Andersen., "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994.

[7] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Aliasing in object-oriented programming," D. Clarke, J. Noble, and T. Wrigstad, Eds. Berlin, Heidelberg: Springer-Verlag, 2013, ch. Alias Analysis for Object-oriented Programs, pp. 196–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=2554511.2554523

[8] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61. [Online]. Available: http://doi.acm.org/10.1145/379605.379665

[9] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/566172.566174

[10] S. Z. Guyer and C. Lin, "Client-driven Pointer Analysis," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 214–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=1760267.1760284

[11] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven Points-to Analysis for Java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 59–76. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094817

[12] M. Sridharan and R. Bodík, "Refinement-based Context-sensitive Points-to Analysis for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 387–400. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134027

[13] L. Shang, X. Xie, and J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 264–274. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259050

[14] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, 2016, pp. 22:1–22:26. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[15] Y. Su, D. Ye, and J. Xue, "Parallel Pointer Analysis with CFL-Reachability," *43rd International Conference on Parallel Processing*, no. ICPP, pp. 451–460, 2014.

[16] X. Zheng and R. Rugina, "Demand-driven Alias Analysis for C," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. New York, NY, USA: ACM, 2008, pp. 197–208. [Online]. Available: http://doi.acm.org/10.1145/1328438.1328464

[17] M. Sridharan, "Discussion in Soot mailing list on demand-driven points-to analyses," 2006, www.sable.mcgill.ca/pipermail/soot-list/2006-January/000477.html.

[18] B. K. Rosen, "Linear Cost is Sometimes Quadratic," in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '81. New York, NY, USA: ACM, 1981, pp. 117–124. [Online]. Available: http://doi.acm.org/10.1145/567532.567545

[19] M. Hind and A. Pioli, "Which Pointer Analysis Should I Use?" in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '00. New York, NY, USA: ACM, 2000, pp. 113–123. [Online]. Available: http://doi.acm.org/10.1145/347324.348916

[20] G. Xu, A. Rountev, and M. Sridharan, "Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis," in *Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP 2009*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 98–122. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_6

[21] J. Späth, K. Ali, and E. Bodden, "Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 48:1–48:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290361

[22] JFreeChart, "A Java Chart Library," 2017, https://sourceforge.net/projects/jfreechart/.

[23] "Source Code: Demand-Driven Context-Sensitive Alias Analysis for Java," 2013, http://web.cs.ucla.edu/~harryxu/tools/files/MayAlias.zip.

[24] Y. Lu, L. Shang, X. Xie, and J. Xue, "An Incremental Points-to Analysis with CFL-Reachability," in *Proceedings of the 22Nd International Conference on Compiler Construction*, ser. CC'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 61–81. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37051-9_4

[25] J. Vedurada and V. K. Nandivada, "Source Code: Batch Alias Analysis (BatchAnalyzer)," 2019, https://archive.softwareheritage.org/browse/origin/https://github.com/jyothivedurada/BatchAnalyzer.git.

[26] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework, booktitle = Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research," ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[27] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *OOPSLA '06*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[28] E. Bodden, A. Sewe, J. Sinschek, M. Mezini, and H. Oueslati, "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders," in *Proceeding of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 241–250.

[29] J. Vedurada and V. K. Nandivada, "Refactoring Opportunities for Replacing Type Code with State and Subclass," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 305–307. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.97

[30] B. G. Ryder, "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages," in *Compiler Construction*, G. Hedin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 126–137.

[31] M. Thakur and V. K. Nandivada, "Compare less, defer more: scaling value-contexts based whole-program heap analyses," in *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*, 2019, pp. 135–146. [Online]. Available: https://doi.org/10.1145/3302516.3307359

[32] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective Context-sensitivity Guided by Impact Pre-analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 475–484. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594318

[33] S. Jeong, M. Jeon, S. Cha, and H. Oh, “Data-driven Context-sensitivity for Points-to Analysis,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 100:1–100:28, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133924

[34] G. M. Rama, R. Komondoor, and H. Sharma, “Refinement in Object-sensitivity Points-to Analysis via Slicing,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 142:1–142:27, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276512

[35] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, “On Abstraction Refinement for Program Analyses in Datalog,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594327

[36] A. Diwan, K. S. McKinley, and J. E. B. Moss, “Type-based Alias Analysis,” in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI ’98. New York, NY, USA: ACM, 1998, pp. 106–117. [Online]. Available: http://doi.acm.org/10.1145/277650.277670