Point-to Analysis for Object-Oriented Language

M Rajasekhara Babu[1], Vivek Dilip Mitkari [1], Karan Thakkar [1], Kirti Barode [1]
[1] School of Computing Science and Engineering, VIT University,
632014 Vellore, India

Abstract—

Pointer analysis is the problem of statically determining the runtime targets of pointer variables in a program. This information has a wide variety of client applications in optimizing compilers and software engineering tools. This paper focuses on precise points-to analysis for Object Oriented Languages (OOL) such as Java, C#, Scala, etc based on Andersen's points-to analysis for C language, which validates the performance of the analysis on a large set of Java programs. Implementation of this analysis is done by using a subset-based approach which is done by using Binary Decision Diagrams (BDDs). This paper first introduces BDDs and operations on BDDs using some simple points-to examples. Then, a complete subset-based points-to algorithm is presented, expressed completely using BDDs and BDD operations.

Keywords: Subset based analysis, Pointer analysis, OOL.

I. INTRODUCTION

This paper requires pointer behavior knowledge to analyze program which uses pointers. Without this knowledge accessing pointer will lead to affect the precision and efficiency of any analysis that requires this information, such as optimizing compilers or a program understanding tools. Pointer analysis is the problem of statically determining the runtime targets of pointer variables in a program. A solution is imprecise if, for any variable, the inferred target set is larger than necessary. Thus, the most imprecise but sound solution has each variable pointing to every other. Obtaining a perfect (i.e. flow-sensitive and context sensitive) solution, however, is undecidable in general [2] and, in practice; obtaining even relatively imprecise information (i.e. flow-insensitive and context-insensitive) is expensive [1].

A software system requires strong support from software engineering tools for program understanding, maintenance, and testing. To determine properties of a program at run-time, various static analysis methods are followed which optimizes compilers and software engineering tools [5]. One of the fundamental static analyses is *points-to analysis*. For OOL, points-to analysis determines the set of objects whose addresses may be stored in a given reference variable or reference object field. The analysis constructs an abstraction for the run-time memory states of the analyzed program by computing such *points-to sets* for variables and fields. This abstraction is typically represented by one or more *points-to graphs*. Points-to analysis enables a variety of other analyses. For example, *side-effect analysis*, which determines the memory locations that may be modified by the execution of a statement, and *def-use analysis*, which identifies pairs of statements that set the value of a memory location and subsequently use the value [5]. Such analyses are needed by compilers to perform well-known optimizations such as code motion and partial redundancy elimination. These analyses are also important in the context of software engineering tools, for example, def-use analysis is needed for program slicing and data-flow-based testing. Points-to analysis is a crucial prerequisite for employing these analyses and optimizations. In this paper we define and evaluate a point-to analysis for OOL which is based on Andersen's points-to analysis for C [6], with all extensions necessary to handle object-oriented features.

At a very high level, one can see this problem as finding the allocation sites that reach a variable in the program. Consider an allocation statement S: a = new A ();. If a variable x is used in some other part of the program, then one would like to know whether x can refer to (point-to) an object allocated at S.

A key problem in developing efficient solvers for the subset-based points-to analysis is that for large programs there are many points-to sets, and each points-to set can become very large. Often, many of these points-to sets are equal or almost equal. In particular, we wanted to examine three aspects of the BDD solution: (a) execution time for the solver, (b) memory usage, and (c) ease of specifying the points-to algorithm using a standard BDD package. In summary, our experience was that BDDs were very effective in all three aspects.

II. RELATED WORK

Pointer analysis is hot cake in the area of compiler optimization. Till 2001, one hundred papers and twelve PhD thesis have been published on pointer analysis at various level of analysis abstraction [7]. (David J. Pearce et al., 2007) proposed Efficient Field-Sensitive Pointer Analysis for C which includes an approach to model indirect struct call and function calls and aggregates for pointer analysis of C. For the first time, an O (v) bound on the time needed for field-sensitive pointer analysis of C is obtained, where v is the number of nodes in the constraint graph. Furthermore, they evaluated its effect on time and precision and these results indicate that field-sensitivity, while offering greater precision, is expensive to compute [8]. Tobias Gutzmann et al. proposed Towards Path-Sensitive Points-to Analysis analyses the basic definitions of pointer analysis, name schema, flow sensitivity, context sensitivity, and SSA (Static Single Assignment). Path sensitive approach was proposed which uses the fact that control flow statements may make branching decisions based upon input variables. A general approach was presented which can be applied to many intermediate representations and its application (with and without SSA form). Implementation for Flow-Insensitive Analysis and Memory SSA based Program Representation is also presented. At last various metrics like abstract metrics, source level metrics, and performance considerations for different type of programs in different languages.Paul Anderson et al. proposed Flow insensitive points-to sets in which pointer analysis is performed. Flow-insensitive analysis assumes that program statements can be executed in any order which consider two algorithms that perform flow and context insensitive points-to analysis. Both algorithms are first considered with structure fields collapsed and then with structure fields expanded. The first is an implementation of Andersen’s algorithm and second an extension of Fahndrich’s algorithm. Both algorithms read in normalized statements that represent the pointer manipulations in the program and comparison is done on the basis of points- to data i.e. collapsed fields, expanded fields and Edge histograms. Maryam Emami et al. proposed Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers.

A new method for computing the points-to information for stack allocated data structures is projected. This method uses the concept of abstract stack locations to capture all possible and definite relationships between accessible stack locations. The method provides context-sensitive inter-procedural information, and it handles general function pointers in an integrated fashion. The points-to information can be used to generate traditional alias pairs, or it can be used directly for numerous other optimization and transformations including pointer replacement and array dependence testing. Rupesh Nasre et al. proposed Prioritizing Constraint Evaluation for Efficient Points-to Analysis where prioritizing approach is projects an inclusion-based point-to analysis. In typical inclusion-based points-to analysis iteratively evaluates constraints and computes points-to solution until a fix point. In each iteration, (i) points-to information is propagated across directed edges in a constraint graph G and (ii) more edges are added by processing the points-to constraints. It is observed that by prioritizing the order of processing the information within each of the above two steps can lead to efficient execution of the points-to analysis. A prioritization framework is developed for implementing prioritized versions of Andersen’s analysis. Ben Hardekopf et al. proposed a Flow-Sensitive Pointer Analysis for Millions of Lines of Code. The typical method for optimizing a flow-sensitive dataflow analysis is to perform a sparse analysis which directly connects variable definitions with their uses, allowing data flow facts to be propagated only to those program locations that need the values. Unfortunately, sparse pointer analysis is problematic because pointer information is required to compute the very def-use information that would enable a sparse analysis. This paper shows how this difficulty can be overcome and how the use of a sparse analysis greatly increases the scalability of flow-sensitive pointer analysis. The key insight behind our technique is to stage the pointer analysis. Auxiliary pointer analysis first computes conservative def-use information, which then enables the primary flow-sensitive analysis to be, performed sparsely using the conservative def-use information. This idea actually defines a family of staged flow-sensitive analyses.

Not only for procedural language but also for object oriented language point-to analysis algorithms have been proposed. Atanas Rountev et al., proposed Points-to Analysis for Java Using Annotated Constraints. authors have define and evaluate a points-to analysis for Java which is based on Andersen's points-to analysis for C , with all extensions necessary to handle object-oriented features. It includes implementation of the analysis is done by using a constraint-based approach which employs annotated inclusion constraints. Constraint annotations allow to model precisely and efficiently the semantics of virtual calls and the flow of values through object fields.

Pointer analysis is hot cake in the area of compiler optimization. Till 2001, one hundred papers and twelve PhD thesis have been published on pointer analysis at various level of analysis abstraction [7]. (David J. Pearce et al., 2007) proposed Efficient Field-Sensitive Pointer Analysis for C which includes an approach to model indirect struct call and function calls and aggregates for pointer analysis of C. For the first time, an O (v) bound on the time needed for field-sensitive pointer analysis of C is obtained, where v is the number of nodes in the constraint graph. Furthermore, they evaluated its effect on time and precision and these results indicate that field-sensitivity, while offering greater precision, is expensive to compute [8]. Tobias Gutzmann et al. proposed Towards Path-Sensitive Points-to Analysis analyses the basic definitions of pointer analysis, name schema, flow sensitivity, context sensitivity, and SSA (Static Single Assignment). Path sensitive approach was proposed which uses the fact that control flow statements may make branching decisions based upon input variables. A general approach was presented which can be applied to many intermediate representations and its application (with and without SSA form). Implementation for Flow-Insensitive Analysis and Memory SSA based Program Representation is also

presented. At last various metrics like abstract metrics, source level metrics, and performance considerations for different type of programs in different languages.Paul Anderson et al. proposed Flow insensitive points-to sets in which pointer analysis is performed. Flow-insensitive analysis assumes that program statements can be executed in any order which consider two algorithms that perform flow and context insensitive points-to analysis. Both algorithms are first considered with structure fields collapsed and then with structure fields expanded. The first is an implementation of Andersen's algorithm and second an extension of Fahndrich's algorithm. Both algorithms read in normalized statements that represent the pointer manipulations in the program and comparison is done on the basis of points- to data i.e. collapsed fields, expanded fields and Edge histograms. Maryam Emami et al. proposed Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers.

A new method for computing the points-to information for stack allocated data structures is projected. This method uses the concept of abstract stack locations to capture all possible and definite relationships between accessible stack locations. The method provides context-sensitive inter-procedural information, and it handles general function pointers in an integrated fashion. The points-to information can be used to generate traditional alias pairs, or it can be used directly for numerous other optimization and transformations including pointer replacement and array dependence testing. Rupesh Nasre et al. proposed Prioritizing Constraint Evaluation for Efficient Points-to Analysis where prioritizing approach is projects an inclusion-based point-to analysis. In typical inclusion-based points-to analysis iteratively evaluates constraints and computes points-to solution until a fix point. In each iteration, (i) points-to information is propagated across directed edges in a constraint graph G and (ii) more edges are added by processing the points-to constraints. It is observed that by prioritizing the order of processing the information within each of the above two steps can lead to efficient execution of the points-to analysis. A prioritization framework is developed for implementing prioritized versions of Andersen's analysis. Ben Hardekopf et al. proposed a Flow-Sensitive Pointer Analysis for Millions of Lines of Code. The typical method for optimizing a flow-sensitive dataflow analysis is to perform a sparse analysis which directly connects variable definitions with their uses, allowing data flow facts to be propagated only to those program locations that need the values. Unfortunately, sparse pointer analysis is problematic because pointer information is required to compute the very def-use information that would enable a sparse analysis. This paper shows how this difficulty can be overcome and how the use of a sparse analysis greatly increases the scalability of flow-sensitive pointer analysis. The key insight behind our technique is to stage the pointer analysis. Auxiliary pointer analysis first computes conservative def-use information, which then enables the primary flow-sensitive analysis to be, performed sparsely using the conservative def-use information. This idea actually defines a family of staged flow-sensitive analyses.

Not only for procedural language but also for object oriented language point-to analysis algorithms have been proposed. Atanas Rountev et al., proposed Points-to Analysis for Java Using Annotated Constraints. authors have define and evaluate a points-to analysis for Java which is based on Andersen's points-to analysis for C , with all extensions necessary to handle object-oriented features. It includes implementation of the analysis is done by using a constraint-based approach which employs annotated inclusion constraints. Constraint annotations allow to model precisely and efficiently the semantics of virtual calls and the flow of values through object fields.

III. MOTIVATION

Modern superscalar and VLIW (Very Long Instruction Word) processors require sufficient Instruction Level Parallelism (ILP) to reach peak utilisation. For this reason, exposing ILP through instruction scheduling and register allocation is a crucial role of the compiler. This task is complicated by the presence of instructions which indirectly reference memory, since their data dependencies are not known. For languages such as C/C++, this problem is particularly acute because pointer variables (the main source of indirect memory references) can target practically every memory location without restriction. Therefore, to achieve maximum pipeline throughput, the compiler must rely on pointer analysis to disambiguate indirect memory references. Pointer analysis is an important enabling technology that can improve the precision and performance of many program analyses by providing precise pointer information.

Finally, pointer analysis finds many other important uses within the compiler. In particular, it often enables traditional optimisations (e.g. common sub expression elimination) to be applied at places which would otherwise be deemed unsafe.

IV. BDD BACKGROUND

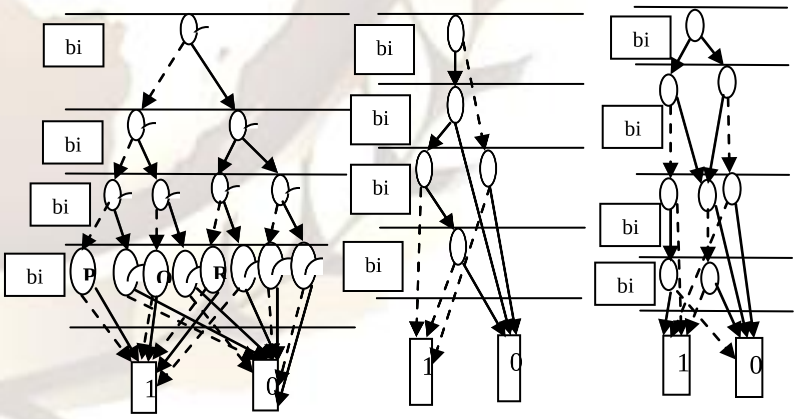
A Binary Decision Diagram (BDD) is a representation of a set of binary strings of length n that is often, equivalently, thought of as a binary-valued function that maps binary strings of length n to 1 if they are in the set or to 0 if they are not. Structurally, a BDD is a rooted directed acyclic graph, with terminal nodes 0 and 1, and where every non-leaf node has two successors: a 0-successor and a 1-successor. As in a binary tree, to determine whether a string is in the set represented by a BDD, one starts at the root node, and proceeds down the BDD by following either the 0- or 1- successor of the current node depending on the value of the bit of the string

being tested. Eventually, one ends up either at 1, indicating that the string is in the set, or at 0 indicating that it is not.
To use a concrete example, consider the program fragment in Figure 1. The points-to relation we would compute for this code is {(a, X), (a, Y), (b, X), (b, Y), (c, X), (c, Y), (c, Z)}, where (a, A) indicates that variable a may point to objects allocated at allocation site X. Using 00 to represent a and X, 01 to represent b and Y, and 10 to represent c and Z, we can encode this points-to relation using the set {0000,0001,0100,0101,1000,1001,1010} .
Figure 2(a) shows an unreduced BDD representing this set where the variables a, b and c are encoded at BDD node levels V0 and V1 and the heap objects X, Y and Z are encoded at the H0 and H1 levels. As a convention, 0-successors are indicated by dotted edges and 1-successors are indicated by solid edges.

```
X: a = new O ();
Y: b = new O ();
Z: c = new O ();
      a = b;
      b = a;
      c = b;
```

Figure 1. Example code fragment

Notice that nodes marked P, Q, and R in Figure 2(a) are at the same level and have the same 0- and 1-successors. This is because they represent the subset {X, Y}, which is shared by all three pro-gram variables. Because they are at the same level and share the same successors, they could be merged into a single node, reducing the size of the BDD. Furthermore, since their two successors are the same (the 1 node), their successor does not depend on the bit being tested, so the nodes could be removed entirely. Simplifying other nodes in this manner, we get the BDD in Figure 2(b). The BDD with the fewest nodes is unique if we maintain a consistent ordering of the nodes; it is called a reduced BDD. When BDDs are used for computation, they are always kept in a reduced form.
In the examples so far, the bits of strings were tested in the order in which they were written. However, any ordering can be used, as long as it is consistent over all strings represented by the BDD.
For example, Figure 2(c) shows the BDD that represents the same relation, but tests the bits in a different order. This BDD requires 8 nodes, rather than 5 nodes as in Figure 2(b). In general, choosing a bit ordering which keeps the BDDs small is very important for efficient computation; however, determining the optimal ordering is NP-hard [4]. BDDs support the usual set operations (union, inter-section, complement, difference) and can be maintained in reduced form during each operation. A binary operation on BDDs X and Y, such as X Y, takes time proportional to the number of nodes in the BDDs representing the operands and result. In the worst case, the number of nodes in the BDD representing the result can be the product of the number of nodes in the two operands, but in most cases, the reduced BDD is much smaller [4].
BuDDy [3] is one of several publicly-available BDD packages. Instead of requiring the programmer to manipulate individual bit positions in BDDs, BuDDy provides an interface for grouping bit positions together. The term domain is used to refer to such a group. In the example in Figure 2, we used the domain V to represent variables, and H to represent pointed-to heap locations.
Another BDD operation is existential quantification. For example, given a points-to relation P H, we can existentially quantify over H to find the set S of variables with non-empty points-to sets: S = {v│ h. (v, h) P}.
The relational product operation implemented in BuDDy composes set intersection with existential quantification, but is implemented more efficiently than these two operations composed. Specifically, *rel-prod*(A, B, V_1) = {(v2, h) │ v1. ((v1, v2) A Λ (v1, h) B)}. To illustrate this with an example, for the code fragment in Figure 1, consider the initial points-to set {(a, X), (b, Y), (c, Z)} (corresponding to the first three lines of code) and the assignment edge set {(b, a), (a, b), (b, c)} (corresponding to the last three lines of code). The pair (a, b) corresponds to the statement b: = a; that is, we write the variables in reverse order, indicating that all allocation sites reaching a also reach b. The initial points-to set is represented in the BDD in Figure 3(a) using the domains V_1 and H_1.



(a) (b)
(c)

Figure 2. BDDs for points-to relation {(a, X), (a, Y), (b, X), (b, Y), (c, X), (c, Y), (c, Z)} (a) Unreduced using ordering V1V0H1H0, (b) reduced using ordering

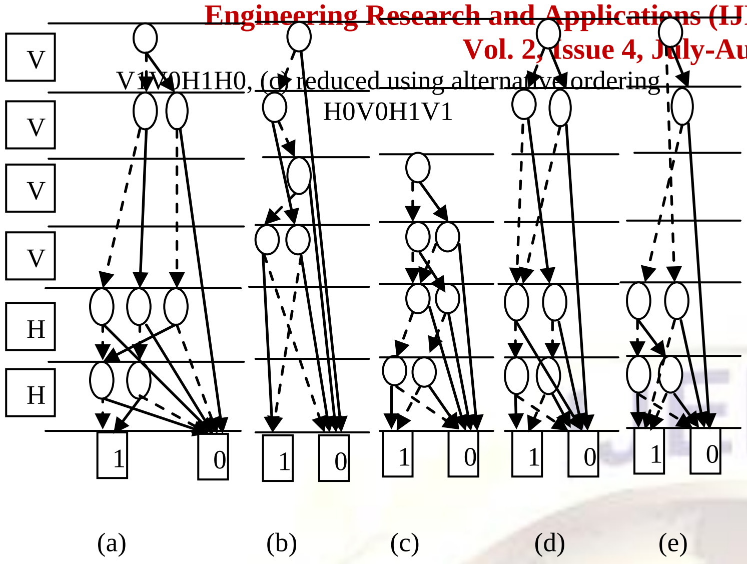


Figure 3. (a) BDD for initial points-to set {(a, X),(b, Y),(c, Z)} (b) BDD for edge set {(a b),(b a), (b c)} (c) result of rel-prod((a),(b),V1) (the points-to set {(a, Y),(b, X),(c, C)}) (d) result of replace((c),V_2ToV$_1$) (e) result of (a) (d) (the points-to set {(a, X),(a, Y),(b, X),(b, Y),(c, Y),(c, Z)}

The edge set contains pairs of variables, so two variable domains (V_1 and V_2) are required to represent it; its representation is shown in Figure 3(b). Given these two BDDs, we can apply the relational product with respect to V_1 to obtain the BDD of the points-to sets after propagation along the edges (Figure 3(c)), using the domains V_2 and H.

The replace operation creates a BDD in which information that was stored in one domain is moved into a different domain. For example, we would like to find the union of the points-to relations in parts (a) and (c) of Figure 3, but the former uses the domains V_1 and H, while the latter uses V_2 and H.

Before finding the union, we are applying the replace operation to (c) to obtain (d), which, like (a), uses domains V_1 and H. We can now find (e) = (a) (d), the points-to set after one step of propagation. If we repeated these steps a second time, we would obtain the final points-to set BDD from Figure 2(b).

Note that it is possible for a BDD for a large set to have fewer nodes than the BDD for a smaller set. In this case, although the points-to set grows from three, to six, to seven pairs, the BDD representing it goes from eight to six to five nodes (see Figures 3(a), 3(e), and 2(b), respectively).

V. POINTS-TO ALGORITHM WITH BDDS

A points-to analysis computes a points-to relation between variables of pointer type and allocation sites. Our analysis is a Java extension of the analysis suggested for C by Andersen [1]. As such, it is both flow-insensitive and context-insensitive. The analysis takes as input constraints modeling four types of statements: allocation, simple assignment, field store, and field load (Figure 4). Pt (l) indicates the points-to set of variable l. l_1 l_2 indicates that l_2 may point to anything that l_1 may point to. Based on a call graph built using class hierarchy analysis [7], we add appropriate assignment edges to model inter-procedural pointer flow through method parameters and return values. We took this approach of generating all the constraints ahead of time because in this first study, we wanted to clearly separate the constraint generator from the solver. In future work, we plan to integrate them more closely, making it possible to experiment with building the call graph on-the-fly as the points-to analysis proceeds. The inference rules shown in Figure 5 are used to compute points-to sets. The basic idea is to apply these rules until a fixed point is reached. The first rule models simple assignments: if l_1 points to O, and is assigned to l_2, then l_2 also points to O. The second rule models field stores: if l points to O_2, and is stored into q.f, then O_1.f also points to O_2 for each O_1 pointed to by q.

a: l := new C	$O_a \in pt(l)$
$l_2 := l_1$	$l_1 \rightarrow l_2$
q.f:= l	l -> q.f
l := p.f	p.f -> l

Figure 4. The four types of pointer statements (constraints).

Similarly, the third rule models field loads: if l is loaded from p.f, and p points to O_1, then l points to any O_2 that O_1.f points to.

$$\frac{l_1 \rightarrow l_2 \quad O \in pt(l_1)}{O \in pt(l_2)} \quad (1)$$

$$\frac{O_2 \in pt(l) l \rightarrow q.f \quad O_1 \in pt(q)}{O_2 \in pt(O_1.f)} \quad (2)$$

$$\frac{p.f \rightarrow l \quad O_1 \in pt(p) \quad O_2 \in pt(O_1.f)}{O_2 \in pt(l)} \quad (3)$$

Figure 5: Inference Rules

This algorithm is still in its infancy and requires huge number of improvements. This algorithm has been improved to extend the support for virtual functions and auto-boxing. The extension to virtual functions can be achieved by integrating the Hierarchical Analysis with Fast Static Analysis. Fast Static Analysis provides excellent results if virtual functions are present. Hence, a hybrid of these two Analytical methods will prove to be helpful in improving the stability of the algorithm. Auto Boxing is used by most of the programmers since it's easy to use and reduces the lines of code. Therefore, compatibility for Auto

Boxing is an essential point that would be taken care of during the improvement.

A. BDD Implementation

The rules presented in Figure 5 apply to elements of points-to (pt) and assignment-edge ( ) relations. In BDDs, we encode them as operations on entire relations, rather than their individual elements. In our algorithm, we map the components of relations onto five BuDDy domains (groups of bit positions).

- FD is a domain representing the set of field signatures.
- V_1 and V_2 are domains of variables of pointer type. We need two such domains in order to represent the relation of two variables.
- H_1 and H_2 are domains of allocation sites. Two are needed, along with the FD domain, in order to represent the pt relation for fields of objects, which contains elements of the form O2 pt (o1. f).

We now describe the most important relations used in the algorithm, along with the domains onto which they are mapped.

- pointsTo V_1 H_1 is the points-to relation for variables, and consists of elements of the form O_2 pt(l).
- fieldPt (H_1 FD) H_2 is the points-to relation for fields of heap objects, and consists of elements of the form O_2 pt(O_1.f).
- edgeSet V_1 V_2 is the relation of simple assignments, and consists of elements of the form l_1 l_2.
- stores V_1 (V_2 FD) is the relation of field stores, and consists of elements of the form l_1 l_2.f .
- loads (V_1 FD) V_2 is the relation of field loads, and consists of elements of the form l_1.f l_2.
- typeFilter V_1 H_1 is a relation which specifies which ob-jects each variable can point-to, based on its declared type. This is used to restrict the points-to sets for variables to the appropriate objects.

The BDD algorithm is given in Figure 6. First, the algorithm loads input constraints and initializes the relations introduced above. The main algorithm consists of an inner loop nested within an outer loop. To make the algorithm easier to understand, we annotated the type of the relations involved in each step of computation. Lines 1.1 to 1.2 implement rule (1). In line 1.1, the edgeSet and pointsTo relations are combined. This rel-prod operation computes relation {(l_2, O) │ l_1.l_1 l_2 Λ O pt (l_1)}, the pre-conditions of rule (1). In line 1.2, the relation is converted to use domains V_1 and H_1 rather than V_2 and H_1, and in line 1.4, it is added into the pointsTo relation. Line 1.3 will be explained later. Lines 2.1 to 2.3 implement rule (2). Line 2.1 computes the intermediate result of the first two pre-conditions: tmpRel1 = {(O_2, q.f) │ l.O_2 pt(l) Λ l q.f}. In line 2.2, tmpRel1 is changed to domains suitable for the next computation. In line 2.3, the resulting relation of all three pre-conditions is computed as {(O_2, O_1.f) │ q.(O_2, q. f) Λ O_1 pt (q)}.

In a similar way, lines 3.1 to 3.3 implement rule (3). Again, the first two pre-conditions are first combined to form a temporary relation (line 3.1), then combined with the results from rule (2) (line 3.2). After changing the result to the appropriate domains (line 3.3), we obtain new points-to pairs to add to the points-to relation. These are merged into the pointsTo set in line 4.2. The algorithm in Figure 6 is very close to the real code of our implementation using the BuDDy package. So far, we have not explained the purpose of lines 1.3 and 4.2. An earlier points-to study [9, 10] showed that static type information is very useful to limit the size of points-to sets by including only allocation sites of a subtype of the declared type of the variable. Lines 1.3 and 4.2 implement this by screening all newly-introduced points-to pairs with a typeFilter relation. This relation is constructed in line 0.3 from three relations read from the input file: the subtype relation between types, the declared type relation between variables and types, and the allocated type relation between allocation sites and type

B. BDD Algorithm for Points-to Analysis

/* --- initialization --- */
/* 0.1 */ load constraints from the input file
/* 0.2 */ initialize pointsTo, edgeSet, loads, and stores
/* 0.3 */ build typeFilter relation
repeat
repeat
/* --- rule 1 --- */
/* 1.1 */ newPt1:[V_2xH_1] = relprod(edgeSet:[V_1xV_2], pointsTo:[V_1xH_1], V_1);
/* 1.2 */ newPt2:[V_1xH_1] = replace(newPt1:[V_2ToV_1], V_2ToV_1);
/* --- apply type filtering and merge into pointsTo relation --- */
/* 1.3 */ newPt3:[V_1xH_1] = newPt2:[V_1xH_1] \ typeFilter:[V_1xH_1];
/* 1.4 */ pointsTo:[V_1xH_1] = pointsTo:[V_1xH_1] [newPt3:[V_1xH_1];
until pointsTo does not change

/* --- rule 2 --- */
/* 2.1 */ tmpRel1:[(V_2xFD)xH_1] = relprod(stores:[V_1x(V_2xFD)], pointsTo:[V_1xH_1], V1);
/* 2.2 */ tmpRel2:[(V_1xFD)xH_2] = replace(tmpRel1:[(V_2xFD)xH_1], V_2ToV_1 & H_1ToH_2);
/* 2.3 */ fieldPt:[(H_1xFD)xH_2] =

```
relprod(tmpRel2:[($V_1$xFD)x$H_2$], pointsTo:[ $V_1$x$H_1$], $V_1$);

/* --- rule 3 --- */
/* 3.1 */ tmpRel3:[($H_1$xFD)x$V_2$] = relprod(loads:[($V_1$xFD)x$V_2$], pointsTo:[$V_1$x$H_1$], $V_1$);
/* 3.2 */ newPt4:[$V_2$x$H_2$] = relprod(tmpRel3:[($H_1$xFD)x$V_2$], fieldPt:[($H_1$xFD)x$H_2$], $H_1$xFD);
/* 3.3 */ newPt5:[ $V_1$x$H_1$] = replace(newPt4:[$V_2$x$H_2$], $V_2$To$V_1$ & $H_2$To$H_1$]);

/* --- rule 4 --- */
/* --- apply type filtering and merge into pointsTo relation --- */
/* 4.1 */ newPt6:[ $V_1$x$H_1$] = newPt5:[ $V_1$x$H_1$] \ typeFilter:[ $V_1$x$H_1$];
/* 4.2 */ pointsTo:[ $V_1$x$H_1$] = pointsTo:[V1xH1] [ newPt6:[$V_1$x$H_1$];
until pointsTo does not change
```

Figure 6: The basic BDD algorithm for points-to analysis

VI. Experiment Results

Here we are considering two factors in choosing a variable ordering: the ordering of domains and interleaving of the variables of different domains. We use the following naming scheme for orderings: when we list several domain names together, their variables are interleaved; when we list domain names separated by underscores, the variables of one domain all come before those of the next. For example, if f_0, f_1,..., f_n are the variables of the domain f_d and v_0 ,v_1,...,v_n are the variables of the domain v_1, the ordering f_dv_1 corresponds to f_0v_0 f_1v_1.... f_nv_n , and $f_d\_v_1$ corresponds to f_0 f_1.... $f_nv_0v_1$....v_n.
Within each domain, the variables are arranged from the most significant bit to the least significant bit, because the more significant bits may not all be used (always 0), and placing them closer to the beginning reduces the BDD size. Using the default ordering $f_dv1v_2h_1h_2$, our BDD solver cannot solve real benchmarks. We investigated the performance bottleneck and found that most of time was spent on the relprod operation for rule (1) (line 1.1 of Figure 6). This operation propagates points-to sets along assignment edges. Since this operation only involves the edgeSet and pointsTo relations, which use the domains v_1, v_2 and h_1, only the arrangement of these three domains affects this operation. We experimented with several arrangements and interleaving of these three key domains. The effect of two different orderings of the domains h_1 and v_1 on the execution time of the rel-prod operation in line 1.1 (on the javac benchmark, with off-line simplification and respecting declared types) is shown in the graph in Figure 7. In the given graph, the x-axis gives the loop iteration number and the y-axis gives the time spent on each iteration of the relprod operation in line 1.1. The solid line corresponds to the case where h_1 comes after v_1 whereas the dotted line corresponds to the case where h_1 comes before v_1. It is observed that the execution time of relprod changes dramatically: with v_1 before h_1, each operation takes less than 0.5s, while with h_1 before v_1, each operation takes about 4.2s on average. Thus, our experiments with other orderings confirm this behavior, and we can conclude that arranging v_1 before h_1 is a good heuristic.

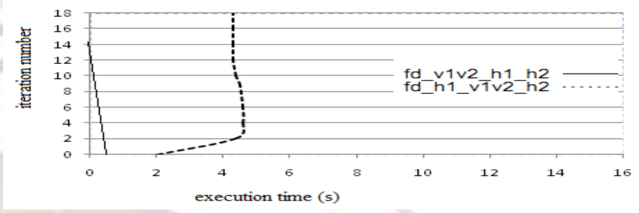


Figure 7. Effect of domain arrangement

VII. Conclusion and Future Work

In this paper, we presented a BDD based points-to analysis that scales very well in terms of time and space, and is very easy to implement using standard BDD packages. The motivation to use BDDs came from the fact that for large programs, the number and size of points-to sets can grow so that even well-tuned traditional representations fail to scale appropriately. BDDs have been shown to work well for large problems in the model checking community, and we wanted to see if it could be applied effectively to the points-to problem. We showed that with the appropriate tuning, a fairly simple algorithm could deliver a solver that was competitive with previously existing solvers and provided a very compact representation of points-to relationships. In our work so far, we concentrated on choosing a good variable ordering and developing the incremental propagation algorithm. It is possible that this could be further improved by introducing some aspects of graph-based solvers into the BDD solver. For example, it would be very interesting to see if efficient BDD algorithms for collapsing strongly connected components [11] would further improve the efficiency of our BDD based points-to algorithm. Another idea which has been suggested for improving the efficiency of BDDs is dynamic variable reordering.

Acknowledgment

The authors would like to thank the School of Computer Science and Engineering, VIT University, for giving them the opportunity to carry out this project and also for providing them with the requisite resources and infrastructure for carrying out the research.

REFERENCES

[1] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, Jan. 1997.

[2] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[3] Jørn Lind-Nielsen. BuDDy, a Binary Decision Diagram Package. Department of Information Technology, Technical University of Denmark, http://www.itu.dk/research/buddy/.

[4] Flow-Sensitive Pointer Analysis for Millions of Lines of Code, by Ben Hardekopf University of California, Calvin Lin The University of Texas at Austin, Semi-sparse flow-sensitive pointer analysis. In Symposium on Principles of Programming Languages (POPL), 2009.

[5] Points-to Analysis for Java Using Annotated Constraints by Atanas Rountev Ana Milanova Barbara G. Ryder Department of Computer Science Rutgers University New Brunswick,NJ08901*f*rountev,milanova,ryder*g*@cs.rutgers.edu.

[6] L. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] Pointer Analysis: Haven't We Solved This Problem Yet? , Michael Hind IBM Watson Research Centre 30 Saw Mill River Road Hawthorne, New York 10532,june 18-19,2001,ACM Journal Snowbird,Utah,USA .

[8] fficient Field-Sensitive Pointer Analysis for C, David J. Pearce, Paul H. J. Kelly, and Chris Hankin. E. Cientifeld, ACMTransactions on Programming Languages and Systems, Vol. 30, No. 1, Article 4, Publication date: Nov. 2007.

[9] Ondˇrej Lhot´ak. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[10] Ondˇrej Lhot´ak and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, Compiler Construction, 12th International Conference, volume 2622 of LNCS, pages 153–169, Warsaw, Poland, April 2003. Springer.

[11] Aiguo Xie and Peter A. Beerel. Implicit enumeration of strongly connected components. In International Conference on Computer-Aided Design, pages 37 – 40, Nov 1999.