Points-to Analysis using BDDs*

Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren and Navindra Umanee
Sable Research Group, School of Computer Science
McGill University
Montreal, Quebec, CANADA H3A 2A7
[berndl,olhotak,fqian,hendren,navindra]@sable.mcgill.ca

ABSTRACT

This paper reports on a new approach to solving a subset-based points-to analysis for Java using Binary Decision Diagrams (BDDs). In the model checking community, BDDs have been shown very effective for representing large sets and solving very large verification problems. Our work shows that BDDs can also be very effective for developing a points-to analysis that is simple to implement and that scales well, in both space and time, to large programs.

The 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. This algorithm is then refined by finding appropriate variable orderings and by making the algorithm propagate sets incrementally, in order to arrive at a very efficient algorithm. Experimental results are given to justify the choice of variable ordering, to demonstrate the improvement due to incrementalization, and to compare the performance of the BDD-based solver to an efficient hand-coded graph-based solver. Finally, based on the results of the BDD-based solver, a variety of BDD-based queries are presented, including the points-to query.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Languages, Experimentation

Keywords

Points-to analysis, binary decision diagrams

1. INTRODUCTION

In this paper, we take a well-known problem from the compiler optimization community, *points-to analysis*, and we show how to solve this problem efficiently using *reduced ordered binary decision diagrams* (ROBDDs)[1] which have been shown to be very effective in the model checking community.

Whole program analyses, such as points-to analysis, require approaches that can scale well to large programs. Two popular approaches to flow-insensitive points-to analysis have been pursued in the past, equality-based approaches like those pioneered by Steensgaard [32], and subset-based approaches like the analysis first suggested by Andersen [1]. The subset-based approaches give more accurate results, but they also lead to greater challenges for efficient implementations [6,10,12,19,25,28,34].[2]

For this paper, we have chosen to implement a subset-based points-to analysis for Java. 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. Several methods of representing them compactly have been studied in the past, including collapsing equivalent variables [10,24] and designing new representations for sets [11,17,18]; the BDD-based approach that we introduce in this paper is another example of such a compact representation.

Since BDDs have been shown to be effective for compactly representing large sets and for solving large state space problems like those generated in model checking, it seemed like an interesting question to see if BDDs could also be used to efficiently solve the points-to problem for Java. 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.

The contributions of this paper include:

- We propose and develop an efficient BDD-based algorithm for subset-based points-to analysis for Java. To our knowledge, we are the first group to successfully use BDDs to solve such an analysis efficiently.
- We provide new insights into how to make the BDD-based implementation efficient in terms of space and time. First, we used a systematic approach to find a good variable ordering for the points-to set. Second, we noted that the al-

*This work was supported, in part, by NSERC and a Tomlinson Graduate Fellowship. Special thanks to Jørn Lind-Nielsen for his publicly-available `BuDDy` package. We would also like to thank the PLDI reviewers who provided several insightful comments.

PLDI'03, June 9–11, 2003, San Diego, California, USA.

[1] In the remainder of this paper we simply refer to BDDs, meaning ROBDDs.

[2] A more detailed description of related work is found in Section 7.

gorithm should propagate the sets incrementally, and presented an incremental version. This general idea of incrementalizing the algorithm may be useful in solving other program analysis problems using BDDs. Third, we found that specifying an analysis using high-level BDD operations allowed us to specify our analysis very compactly and it was very simple to experiment with a wide variety of algorithms. Our source code (available on our web page: `http://www.sable.mcgill.ca/bdd/`) contains many different variations that can be enabled by switches.

- We experimentally validated the BDD-based approach by comparing its performance to a previously existing efficient solver. For small problem sizes, we found that the time and space requirements are similar, but for larger problems, the BDD-based approach requires less memory, and scales better.

- Although we initially intended to compute only points-to sets, we found that the BDD approach leads to a solution that can be used to answer a variety of queries, of which the points-to query is only one. We suggest several possible other queries. In future work we plan to develop this aspect of our work further.

The rest of this paper is organized as follows. In Section 2 we provide an introduction to BDDs and operations on BDDs using small examples based on the points-to problem. Given this introductory material, we then introduce our points-to algorithm and its implementation using BDDs in Section 3. Then, in Section 4, we show how to improve the performance of the algorithm by choosing the correct variable ordering and making the algorithm incremental. In Section 5 we give experimental results for our best BDD algorithm and compare its performance to a hand-coded and optimized solver based on SPARK. In Section 6 we discuss possible applications for the results of our algorithm, which includes answering points-to queries. Finally, Section 7 gives a discussion of related work and Section 8 gives conclusions and future work.

2. 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 $\boxed{0}$ and $\boxed{1}$, and where every non-leaf node has two successors: a *0-successor* and a *1-successor*. As in a binary trie, 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 $\boxed{1}$, indicating that the string is in the set, or at $\boxed{0}$ 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 $\{(a,A),(a,B),(b,A),(b,B),(c,A),(c,B),(c,C)\}$, where (a,A) indicates that variable a may point to objects allocated at allocation site A. Using 00 to represent a and A, 01 to represent b and B, and 10 to represent c and C, 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 V_0 and V_1 and the heap objects A, B and C are encoded at the H_0 and H_1 levels. As a convention, 0-successors are indicated by dotted edges and 1-successors are indicated by solid edges.

```
A: a = new O();
B: b = new O();
C: c = new O();
   a = b;
   b = a;
   c = b;
```

Figure 1: Example code fragment.

Notice that nodes marked x, y, and z in Figure 2(a) are at the same level and have the same 0- and 1-successors. This is because they represent the subset $\{A,B\}$, which is shared by all three program 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 $\boxed{1}$ 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 [23]. BDDs support the usual set operations (union, intersection, complement, difference) and can be maintained in reduced form during each operation. A binary operation on BDDs A and B, such as $A \cup B$, 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 [23].

`BuDDy` [20] 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 $P \subseteq V \times H$, we can existentially quantify over H to find the set S of variables with non-empty points-to sets: $S = \{v \mid \exists h.(v,h) \in P\}$.

The *relational product* operation implemented in `BuDDy` composes set intersection with existential quantification, but is implemented more efficiently than these two operations composed. Specifically, $relprod(X,Y,V1) = \{(v_2,h) \mid \exists v_1.((v_1,v_2) \in X \wedge (v_1,h) \in Y)\}$. To illustrate this with an example, for the code fragment in Figure 1, consider the initial points-to set $\{(a,A),(b,B),(c,C)\}$ (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 $V1$ and H.

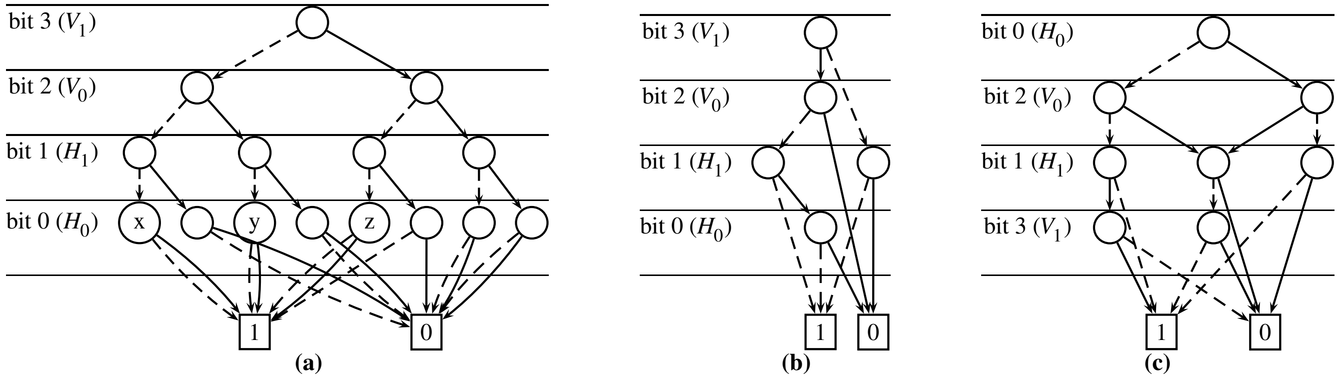


Figure 2: BDDs for points-to relation $\{(a,A),(a,B),(b,A),(b,B),(c,A),(c,B),(c,C)\}$ (a) unreduced using ordering $V_1V_0H_1H_0$, (b) reduced using ordering $V_1V_0H_1H_0$, (c) reduced using alternative ordering $H_0V_0H_1V_1$

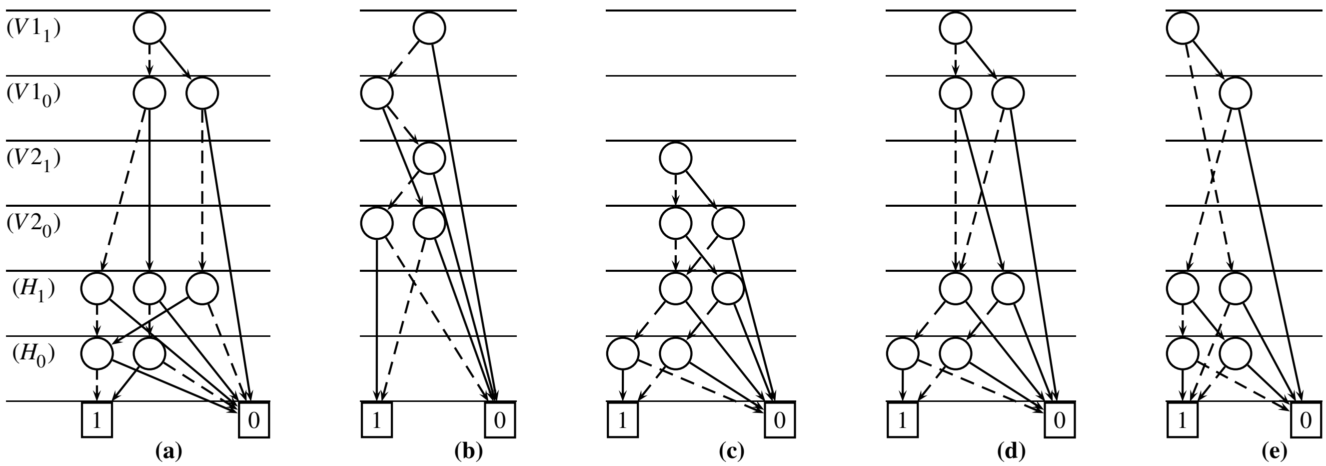


Figure 3: (a) BDD for initial points-to set $\{(a,A),(b,B),(c,C)\}$ (b) BDD for edge set $\{(a \to b),(b \to a),(b \to c)\}$ (c) result of relprod((a),(b),V1) (the points-to set $\{(a,B),(b,A),(c,B)\}$) (d) result of replace((c),V2ToV1) (e) result of (a)$\cup$(d) (the points-to set $\{(a,A),(a,B),(b,A),(b,B),(c,B),(c,C)\}$

The edge set contains pairs of variables, so *two* variable domains ($V1$ and $V2$) 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 $V1$ to obtain the BDD of the points-to sets after propagation along the edges (Figure 3(c)), using the domains $V2$ 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 $V1$ and H, while the latter uses $V2$ and H.

Before finding the union, we applying the replace operation to (c) to obtain (d), which, like (a), uses domains $V1$ and H. We can now find (e)=(a)$\cup$(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).

3. 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- and context-insensitive. The analysis takes as input constraints modelling four types of statements: allocation, simple assignment, field store, and field load (Figure 4). $pt(l)$ indicates the points-to set of variable l. $l_1 \to l_2$ 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

$a : l := new\ C$	$o_a \in pt(l)$
$l_2 := l_1$	$l_1 \to l_2$
$q.f := l$	$l \to q.f$
$l := p.f$	$p.f \to l$

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

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. 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.

$$\frac{l_1 \to l_2 \quad o \in pt(l_1)}{o \in pt(l_2)} \tag{1}$$

$$\frac{o_2 \in pt(l) \quad l \to q.f \quad o_1 \in pt(q)}{o_2 \in pt(o_1.f)} \tag{2}$$

$$\frac{p.f \to l \quad o_1 \in pt(p) \quad o_2 \in pt(o_1.f)}{o_2 \in pt(l)} \tag{3}$$

Figure 5: Inference rules

3.1 BDD Implementation

The rules presented in Figure 5 apply to elements of points-to (pt) and assignment-edge ($\to$) 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.
- $V1$ and $V2$ are domains of variables of pointer type. We need two such domains in order to represent the $\to$ relation of two variables.
- $H1$ and $H2$ 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 $o_2 \in pt(o_1.f)$.

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

- $pointsTo \subseteq V1 \times H1$ is the points-to relation for variables, and consists of elements of the form $o \in pt(l)$.
- $fieldPt \subseteq (H1 \times FD) \times H2$ is the points-to relation for fields of heap objects, and consists of elements of the form $o_2 \in pt(o_1.f)$.
- $edgeSet \subseteq V1 \times V2$ is the relation of simple assignments, and consists of elements of the form $l_1 \to l_2$.
- $stores \subseteq V1 \times (V2 \times FD)$ is the relation of field stores, and consists of elements of the form $l_1 \to l_2.f$.
- $loads \subseteq (V1 \times FD) \times V2$ is the relation of field loads, and consists of elements of the form $l_1.f \to l_2$.
- $typeFilter \subseteq V1 \times H1$ is a relation which specifies which objects 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 relprod operation computes the relation $\{(l_2,o) \mid \exists l_1.l_1 \to l_2 \wedge o \in pt(l_1)\}$, the pre-conditions of rule (1). In line 1.2, the relation is converted to use domains $V1$ and $H1$ rather than $V2$ and $H1$, 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: $tmpRel1 = \{(o_2,q.f) \mid \exists l.o_2 \in pt(l) \wedge l \to q.f\}$. 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 $\{(o_2,o_1.f) \mid \exists q.(o_2,q.f) \wedge o_1 \in pt(q)\}$.

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 [17, 18] 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 types.

4. PERFORMANCE TUNING

As we have seen in section 2, different variable orderings result in different sizes of the BDD for the same set. By default, `BuDDy` interleaves the variables of all domains; this tends to be a good ordering for transition systems in model checking. For our points-to problem set, however, the default ordering turns out to only work on toy problems, and was very slow on real benchmarks. This lead us to explore a variety of different variable orderings by rearranging and interleaving domains. In practice, different orderings yield dramatically different performance. The best ordering we encountered gives impressive results even without further optimizations. When, in addition, we applied incrementalization, the performance of the BDD solver became very competitive compared to a carefully hand-coded solver. Before introducing variable orderings and optimizations, we first describe the experimental setup on which our performance profiling and tuning were done.

4.1 Experimental Setup

We selected benchmarks from the SPECjvm98 [31] suite, SPECjbb2000 [30] and three other large benchmarks: `sablecc-j`, `soot-c` and `jedit`. Table 1 shows the description of each benchmark. `Sablecc-j` is a parser generator written in

```
/* --- 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:[V2xH1] = relprod(edgeSet:[V1xV2], pointsTo:[V1xH1], V1);
    /* 1.2 */ newPt2:[V1xH1] = replace(newPt1:[V2ToV1], V2ToV1);

    /* --- apply type filtering and merge into pointsTo relation  --- */
    /* 1.3 */ newPt3:[V1xH1] = newPt2:[V1xH1] ∩ typeFilter:[V1xH1];
    /* 1.4 */ pointsTo:[V1xH1] = pointsTo:[V1xH1] ∪ newPt3:[V1xH1];
  until pointsTo does not change

  /* --- rule 2 --- */
  /* 2.1 */ tmpRel1:[(V2xFD)xH1] = relprod(stores:[V1x(V2xFD)], pointsTo:[V1xH1], V1);
  /* 2.2 */ tmpRel2:[(V1xFD)xH2] = replace(tmpRel1:[(V2xFD)xH1], V2ToV1 & H1ToH2);
  /* 2.3 */ fieldPt:[(H1xFD)xH2] = relprod(tmpRel2:[(V1xFD)xH2], pointsTo:[V1xH1], V1);

  /* --- rule 3 --- */
  /* 3.1 */ tmpRel3:[(H1xFD)xV2] = relprod(loads:[(V1xFD)xV2], pointsTo:[V1xH1], V1);
  /* 3.2 */ newPt4:[V2xH2] = relprod(tmpRel3:[(H1xFD)xV2], fieldPt:[(H1xFD)xH2], H1xFD);
  /* 3.3 */ newPt5:[V1xH1] = replace(newPt4:[V2xH2], V2ToV1 & H2ToH1]);

  /* --- apply type filtering and merge into pointsTo relation --- */
  /* 4.1 */ newPt6:[V1xH1] = newPt5:[V1xH1] ∩ typeFilter:[V1xH1];
  /* 4.2 */ pointsTo:[V1xH1] = pointsTo:[V1xH1] ∪ newPt6:[V1xH1];
until pointsTo does not change
```

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

benchmark	description
compress	Modified Lempel-Ziv method (LZW).
db	Performs multiple database functions on memory resident database.
raytrace	A raytracer that works on a scene depicting a dinosaur.
mpegaudio	Decompresses audio files that conform to the ISO MPEG Layer-3 audio specification.
jack	A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS).
jess	Java Expert Shell System.
sablecc-j	An object-oriented parser generator written in Java.
jbb2000	A Java program emulating a 3-tier system with emphasis on the middle tier.
javac	The Java compiler from the JDK 1.0.2.
soot-c	A bytecode to bytecode optimization and annotation framework.
jedit	A full-featured editor written in Java.

Table 1: Benchmark description

Java, and `soot-c` is a bytecode transformation framework. Both are non-trivial Java applications, and are publicly-available in the Ashes [2] suite. `Jedit` [15] is a full-featured editor written in Java.

We generated the constraints for our BDD-based solver using the SPARK points-to analysis framework [17, 18]. Constraints were generated for a field-sensitive analysis (using a separate points-to set for each field of the objects allocated at each allocation site). The call graph used for interprocedural flow of pointers was constructed using class hierarchy analysis. Effects of native methods were considered, as supported by SPARK.

The raw points-to constraints generated from an input program can either be fed directly to a solver as input, or they can first be simplified off-line by substituting a single variable for groups of variables known to have the same points-to set [24]. This results in a smaller set of constraints for an equivalent problem. We used the SPARK framework to generate both unsimplified and simplified versions of the constraints as input to our BDD solver. We denote a simplified set of constraints with the letter *s*, and an unsimplified set of constraints with *ns*.

A points-to analysis solver for a typed language such as Java has two reasonable options for dealing with the declared types of variables: it can solve the points-to constraints first, and restrict the points-to sets to subtypes of the declared type afterwards [25, 33]; alternatively, it can remove objects of incompatible type as the points-to analysis proceeds [17, 18, 34]. Both our BDD solver and the SPARK solver support both variations. We denote a solver that respects declared types throughout the analysis with the letter *t*, and one that ignores them until the end with *nt*.

The two options for simplification and two options for handling of types result in four combinations, all four of which have been used in related work. In this study, we focus on three of them: *(s/t)*, *(ns/t)*, and *(ns/nt)*. We stress that the *same* sets of constraints were used as input to both our BDD solver and the SPARK solver.

The BDD Solver is written in C++ and uses the `BuDDy` 2.1 C++ interface compiled with GCC 2.95.4 at -O3. The `BuDDy` package has a built-in reference counting mark-and-compact garbage collector for recycling BDD nodes. Whenever the proportion of free nodes is less than a threshold (20% by default), the kernel increases the node table size. In our experiments for performance measurement, we used a heap size of 160M. All our experimental data were collected on a 1.80 GHz Pentium 4 with 1 GB of memory running Linux 2.4.18.

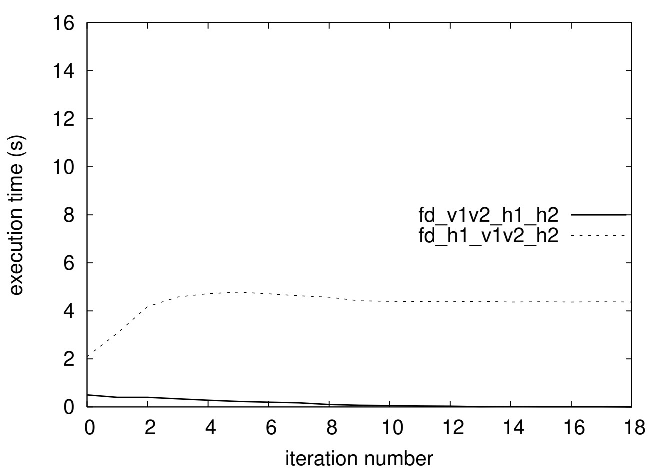
4.2 Variable Ordering

In this section, we describe the path leading us to find efficient orderings and empirically compare several representative orderings.

We consider 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 $f_0, f_1, \ldots, f_n$ are the variables of the domain fd and $v_0, v_1, \ldots, v_n$ are the variables of the domain v1, the ordering fdv1 corresponds to $f_0 v_0 f_1 v_1 \ldots f_n v_n$, and fd_v1 corresponds to $f_0 f_1 \ldots f_n v_0 v_1 \ldots v_n$. 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 fdv1v2h1h2, 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 v1, v2 and h1, only the arrangement of these three domains affects this operation. We experimented with several arrangements and interleavings of these three key domains.

The graph in Figure 7 shows the effect of two different orderings of the domains h1 and v1 on the execution time of the *relprod* operation in line 1.1 (on the `javac` benchmark, with off-line simplification and respecting declared types). 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 h1 comes after v1 whereas the dotted line corresponds to the case where h1 comes before v1. Note that the execution time of *relprod* changes dramatically: with v1 before h1, each operation takes less than 0.5s, while with h1 before v1, each operation takes about 4.2s on average. Our experiments with other orderings confirm this behavior, and we conclude that arranging v1 before h1 is a good heuristic.



the execution time of *relprod* operation

Figure 7: Effect of domain arrangement

We also measured the size of the *pointsTo* relation for these two orderings expecting the slower ordering to correspond to a larger relation. Surprisingly, we did not find this to be the case, both orderings gave similar sizes. This indicates that the *relprod* operation in `BuDDy` is sensitive to not only the size of operands but also the variable ordering.

In other experiments, we found that arranging v1 before v2 for the *edgeSet* relation yields better performance, but this has a much smaller effect than rearranging v1 and h1. One possible explanation is that the *edgeSet* relation (using the domains v1 and v2) remains constant during the computation, while the *pointsTo* relation (using domains v1 and h1) is repeatedly recomputed; therefore, the order of v1 and h1 affects the computation more.

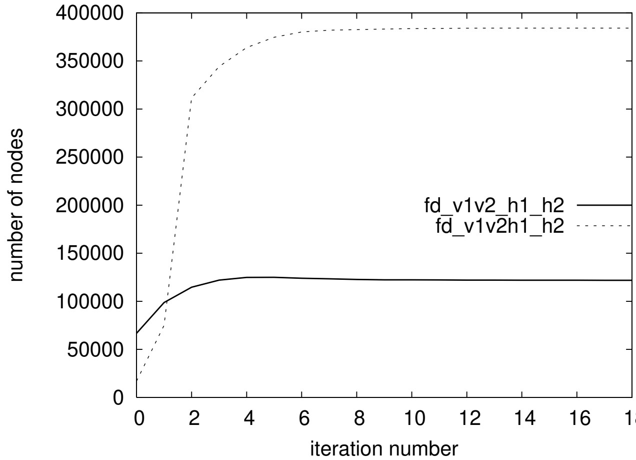
After determining the order in which to arrange domains, we looked at the effect of interleaving them. Again, we only considered v1, v2, and h1, since these are the domains involved in the most expensive operation. Figure 8 shows the effect on the BDD size of the *pointsTo* relation and on the execution time of line 1.1 in each iteration. When v1 and h1 are interleaved, the BDD for the initial points-to relation is much smaller than when they are placed one after the other. However, as the analysis proceeds, the BDD with the interleaved ordering grows to about five times the size of the BDD with v1 before h1. This is because the points-to sets begin to grow as the analysis proceeds, but it appears that the latter BDD is able to exploit their regularity and remain small. The size increase in the BDD with the interleaved ordering degrades the performance significantly. Thus, a good heuristic is to place v1 before h1, and not interleave them.

Table 2 summarizes the performance of the BDD solver with four representative orderings. Column (a) corresponds to the default ordering used by `BuDDy`; this ordering cannot solve real benchmarks in a reasonable time. Column (b) is another example of a bad ordering, with h1 before v1. This ordering already allows the solver to finish on small inputs. The last two columns show the performance when using a good domain arrangement, without interleaving v1 and h1. The performance improvement is dramatic. The difference between last two columns shows the effect of interleaving v1 and v2. This effect is much less significant. The BDD for the *edgeSet* relation is smaller when v1 and v2 are interleaved, and we observed fewer garbage collections. The *pointsTo* relation has the same size with either ordering. On small inputs *(s/t)*, the two orderings yield comparable performance. On large problem sets *(ns/nt)*, interleaving v1 and v2 gives much better performance.

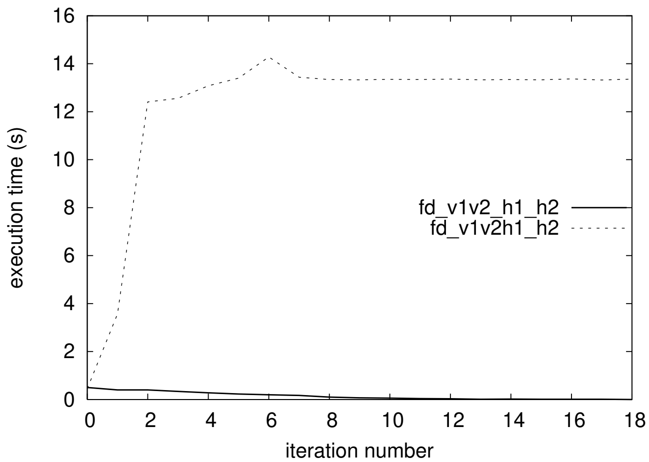
(a) - fdv1v2h1h2 (b) - h1_v1v2_fd_h2
(c) - fd_v1v2_h1_h2 (d) - fd_v1_v2_h1_h2

benchmark	(a)	(b)	(c)	(d)
compress (s/t)	6420s	996s	21s	19s
compress (ns/t)	N/C	4200s	53s	84s
compress (ns/nt)	N/C	8280s	145s	228s
javac (s/t)	9360s	1203s	23s	24s
javac (ns/t)	N/C	4920s	62s	104s
javac (ns/nt)	N/C	10140s	167s	286s
sablecc-j (s/t)	9960s	1388s	22s	23s
sablecc-j (ns/t)	N/C	5700s	63s	111s
sablecc-j (ns/nt)	N/C	9480s	158s	269s
jedit (s/t)	N/C	2460s	36s	35s
jedit (ns/t)	N/C	N/C	112s	358s
jedit (ns/nt)	N/C	N/C	336s	784s

Table 2: Effect of variable ordering on performance (N/C means the solver does not complete the run in 4 hours.)



(a) the BDD size of *pointsTo* relation



(b) the execution time of *relprod* operation

Figure 8: Effect of interleaving domains

4.3 Incrementalization

So far, we have seen that ordering has a huge effect on the performance of the basic BDD solver. The empirical study and analysis lead us to find good orderings that yield reasonable performance in our basic algorithm. However, this is not the end of the story. We apply one further optimization on our basic algorithm, which improves the performance to compete with a highly efficient, hand-coded solver.

Whenever the *pointsTo* relation changes, the solver propagates points-to pairs by repeating the *relprod* operation in line 1.1 until it reaches a fixed point. The execution time of the *relprod* operation is proportional to the sizes of the BDDs being combined, in this case *pointsTo* and *edgeSet*. In each iteration, we propagate *all* points-to sets along all edges, even though most points-to sets have already been propagated in previous iterations. This leads us to an additional optimization, incrementalization.

The idea is to propagate, in each iteration, only the part of each points-to set that has been newly introduced since the last iteration (the old part has already been propagated). Figure 9 shows the replacement for rule (1) of the algorithm. Notice that the *pointsTo* operand of *relprod* in line 1.1 has been replaced with a *newPointsTo* relation, which holds only newly introduced points-to pairs. A new line 1.3 has been added, in which any old points-to pairs that have already been propagated (and are therefore present in the *pointsTo* relation) are removed from the *newPointsTo* relation. This optimization keeps the *newPointsTo* relation small (both in terms of set size, and number of BDD nodes), greatly speeding up the *relprod* operation.

benchmark	fd_v1v2_h1_h2		fd_v1_v2_h1_h2	
	non-inc	inc	non-inc	inc
compress (s/t)	20.63	11.72	19.07	9.80
compress (ns/t)	54.46	26.83	83.63	19.66
compress (ns/nt)	145.33	71.55	228.21	58.58
javac (s/t)	22.62	14.83	23.89	10.83
javac (ns/t)	62.35	30.55	103.52	23.14
javac (ns/nt)	166.66	80.04	285.65	65.46
sablecc-j (s/t)	21.90	14.00	23.10	10.60
sablecc-j (ns/t)	63.43	30.05	110.87	22.86
sablecc-j (ns/nt)	158.33	76.53	269.30	63.82
jedit (s/t)	35.92	20.11	35.43	15.60
jedit (ns/t)	112.47	47.53	357.97	35.29
jedit (ns/nt)	336.18	150.72	783.92	120.53

Table 3: Analysis time improvement due to incrementalization

In a similar way, we also applied incrementalization to the other rules. The details are omitted here; the full incrementalized algorithm is given in Appendix A and the full source code can be found on our web site `http://www.sable.mcgill.ca/bdd/`.

Table 3 shows the improvements in analysis times due to incrementalization on the two good variable orderings that we identified earlier. For the ordering fd_v1v2_h1_h2, incrementalization improves the performance almost 100% for all input sizes. With the ordering fd_v1_v2_h1_h2, the improvement is even more dramatic, and becomes more significant when the problem becomes larger. On `jedit` *(ns/t)*, incrementalization makes the solver almost 10 times faster. Combined with incrementalization, the ordering fd_v1_v2_h1_h2 outperforms fd_v1v2_h1_h2 on all of the benchmarks.

By finding a good variable ordering, and by incrementalizing the algorithm, we have significantly improved its performance, to the point that it competes with hand-coded points-to solvers. In fact, as we show in the next section, for large problems, the BDD algorithm scales well and produces extremely compact encodings of the points-to sets.

5. FULL EXPERIMENTAL RESULTS

We introduced our benchmarks and experimental setup in section 4.1. We also presented the performance of the BDD solver on four benchmarks during the tuning process. In this section, we present and discuss the performance of our best variable ordering for BDDs compared to an efficient, hand-coded solver, SPARK [17, 18], in three aspects: time, memory requirements, and scalability. SPARK includes several different solving algorithms; we used the incremental worklist propagation algorithm, the fastest one.

We are interested not only in scalability in terms of the size of the program being analyzed, but also in the dependence of the solver on techniques that reduce the size of the problem being solved, such as respecting declared types and off-line variable substitution. This is because these techniques may not always be applicable. For example, if we were analyzing a program written in C rather than Java, we could not rely on declared types. Even when analyzing Java, declared types are not always used [25] because they may be

```
newPointsTo = pointsTo;
repeat
  /* --- rule 1 --- */
  /* 1.1 */ newPt1:[V2xH1] = relprod(edgeSet:[V1xV2], newPointsTo:[V1xH1], V1);
  /* 1.2 */ newPt2:[V1xH1] = replace(newPt1:[V2ToV1], V2ToV1);

  /* --- remove old (already propagated) points-to pairs  --- */
  /* 1.3 */ newPt3:[V1xH1] = newPt2:[V1xH1] \ pointsTo:[V1xH1];

  /* --- apply type filtering and merge into pointsTo relation  --- */
  /* 1.4 */ newPointsTo:[V1xH1] = newPt3:[V1xH1] ∩ typeFilter:[V1xH1];
  /* 1.5 */ pointsTo:[V1xH1] = pointsTo:[V1xH1] ∪ newPointsTo:[V1xH1];
until pointsTo does not change
```

Figure 9: Incremental modification to rule (1) of algorithm

inconvenient to represent in the specific solver being implemented. Off-line variable substitution is applicable only when all the constraints are available before the analysis begins; we could not use it if we were computing the call graph on-the-fly as the points-to analysis proceeds.

Table 4 presents our benchmarks ordered by the size of the problem sets and grouped under the headings (s/t), (ns/t) and (ns/nt), where *s* denotes simplification of the set of constraints, while *ns* denotes no simplification, and *t* denotes use of type information during propagation, whereas *nt* denotes no use of type information during propagation. The column labelled *constraints* gives the size of the **input** in terms of the number of constraints, including allocation sites, direct assignments, and field loads and stores. Without simplification, we see that numbers of constraints for the benchmarks range from 316K to 433K. The *set size* column indicates the sum of the sizes of the computed points-to sets across all variables, and is an indication of the size of the solution (the **output**).

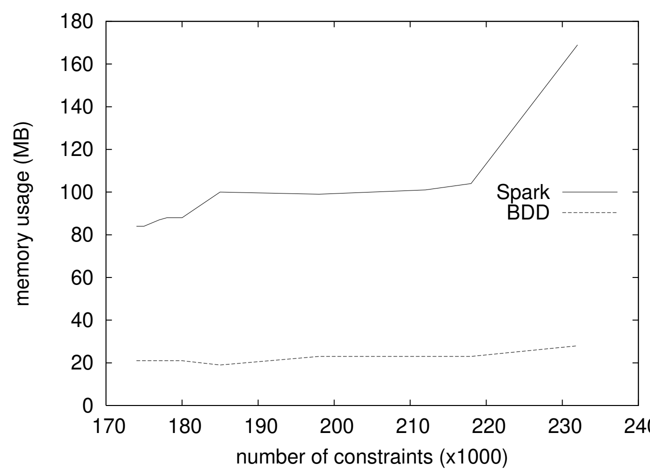


Figure 10: Memory usage vs. number of constraints *(s/t)*

5.1 Performance of BDDs

For small problem sets, we find that our BDD solver is very competitive in terms of solving time, and even begins to beat SPARK when solving `jedit`, our largest benchmark. Further, in terms of memory usage, the BDD solver is a clear winner. Figure 10 plots the memory usage versus the number of input constraints for the *(s/t)* case, where the constraint set has been simplified and the type information is used. Note that not only is the absolute space used much lower for the BDD solver, but also the space used for the BDD solver scales very well as the number of constraints increases.

The difference in space usage widens when we consider the other cases, *(ns/t)* and *(ns/nt)*, where the number of input constraints are larger. As indicated by the middle section of Table 4, without simplification but using type information *(ns/t)*, the BDD solver scales gracefully to a mere 38 MB of memory usage in the worst case, whereas the traditional solver requires up to 244 MB of RAM.

The final section of Table 4 shows the results when type information is not used. Note that the size of output sets increases dramatically, they are about an order of magnitude larger than when types are used. This makes it clear that solvers using an explicit representation of points-to sets will have difficulty scaling if type information is not used, because the total size of the points-to sets becomes up to 356 million elements. This number includes only points-to sets corresponding to variables; the points-to sets corresponding to fields of heap objects (the *fieldsPt* relation in our algorithm) are more than an order of magnitude larger. Even a very efficient set representation using a single bit per set element would require a huge amount of memory to store these sets. In fact, the SPARK solver uses such an encoding, and it ran out of memory on all of the problems which ignored declared types, even on a larger machine with 2 GB of memory. However, when we look at the results for the BDD solver we see that even though the set size has increased by about a factor of 10 over the *(ns/t)* case, the BDD memory usage increases by less than a factor of 2, and solves all problems in less than 66 MB. This demonstrates the ability of BDDs to take advantage of the sharing of large sets.

Overall, we see that both solvers are efficient on small problem sets, and that the hand-coded solver is good at handling inputs with type information. As the problem sets get larger, however, the BDD solution shows a remarkable ability to scale well and handle large points-to sets by exploiting regularity in the sets to keep the representation small. Techniques in which declared types are not used to limit the size of points-to sets, which have been used in related work [25, 33], would not be able to scale to this size of problem; however, the BDDs do.

5.2 Notes on Measuring Memory Usage

As previously noted, for the timing run of the BDD solver, we allotted a heap of 160 MB to reduce the frequency of garbage collections. However, the solver never requires this full amount during

benchmark	const-raints 10^3	set size 10^6	BDD time (s)	BDD mem (MB)	SPARK time (s)	SPARK mem (MB)
(s/t)						
compress	174	6.7	10	21	8	84
db	174	6.8	10	21	8	84
raytrace	175	6.7	10	21	8	84
jack	177	7.1	10	21	8	87
mpegaudio	178	7.0	10	21	8	88
jess	180	7.2	10	21	8	88
jbb2000	185	7.8	10	19	9	100
javac	198	8.0	11	23	10	99
sablecc-j	212	7.3	11	23	8	101
soot-c	218	9.9	12	23	10	104
jedit	232	12.2	16	28	19	169
(ns/t)						
compress	316	18.3	20	29	11	127
db	317	18.4	20	28	11	128
raytrace	318	18.3	22	29	11	129
jack	325	19.2	22	29	12	132
mpegaudio	325	19.1	23	30	13	134
jess	330	19.8	24	29	12	136
jbb2000	342	20.9	22	23	13	159
javac	366	21.0	23	31	14	148
sablecc-j	393	19.9	23	31	14	158
soot-c	397	24.9	26	33	15	162
jedit	433	35.1	35	38	35	244
(ns/nt)						
compress	316	163.3	59	38	oom	oom
db	317	163.9	59	38	oom	oom
raytrace	318	163.4	59	38	oom	oom
jack	325	171.5	59	38	oom	oom
mpegaudio	325	171.2	61	39	oom	oom
jess	330	183.3	61	39	oom	oom
jbb2000	342	206.7	70	42	oom	oom
javac	366	199.7	65	40	oom	oom
sablecc-j	393	196.3	64	41	oom	oom
soot-c	397	228.4	66	43	oom	oom
jedit	433	344.9	121	66	oom	oom

Table 4: Performance and live data of BDD solver.
The value "oom" indicates that the solver ran out of memory, even on a machine with 2 GB.

the computation.

To measure the actual memory usage, we started `BuDDy` with 23 MB as the initial size of the node table and a 2.4 MB cache size. Whenever less than 20% of memory was left, `BuDDy` increased the size of the node table by 2.4 MB. This allowed us to measure the maximum live set size to within 2.4 MB. Note that the actual memory allocated is up to 20% higher than this number, because `BuDDy` always maintains 20% of unused nodes for future use.

In SPARK, the memory requirements increase monotonically as the analysis proceeds, so we simply report the final live set size after a garbage collection at the end of the analysis. This comparison can only be used as a rough reference, since Java and C++ have different object models and memory management. However, we are measuring only the size of the data structures used by the points-to analysis, excluding any structures introduced by the language-specific run-time system. The memory requirements should therefore be at least roughly comparable, in spite of the differences between Java and C++.

6. APPLICATIONS

Section 5 shows that the BDD encoding scheme allows the points-to problem to be solved quickly and represents the points-to relation compactly. A compiler has several options for accessing this information. One option is to extract the entire relation into an explicit representation of the points-to sets. The time to enumerate the set of satisfying bit-vectors for a BDD X is nearly linear to the number of solutions[3]. Alternatively, it can selectively extract only the points-to set of a variable, or those of a specific set of variables, on demand. A third, interesting option is to minimize or even eliminate the need to extract an explicit representation, but rather use the BDD representation and operations for further computation. The size of the explicit representation of the relation stored in a BDD may be quite large; by encoding part or all of the subsequent set processing with BDD operations, we can avoid constructing an explicit representation of this possibly large relation.

The most common use of points-to information is to determine whether two heap references $p.f$ and $q.f$ could refer to same location, which reduces to checking whether p and q could be aliased. A demand-driven way to solve this is to extract the points-to set of $pt(p)$ and $pt(q)$ by using the standard BDD operation, *restrict*, and check if the intersection of two sets is not empty (an empty set in BDD equals to the `bdd_false` constant):

```
pt(p) = restrict(pointsTo, BDD(p), V1);
pt(q) = restrict(pointsTo, BDD(q), V1);
alias(p,q) = (pt(p) ∩ pt(q) ≠ bdd_false);
```

Another common use of points-to information is virtual method call resolution. To resolve a method call site, a compiler needs a set of possible types of the receiver. This can be determined by checking the types of the objects found in the points-to set of the receiver. As the object types are encoded in a BDD relation, we can find the sets of possible receiver types for all receivers using just one relational product operation. Thus we can find all the types for all the receivers in just one step.

```
varTypes = relprod(pointsTo, objectType, H1)
```

The implicit BDD encoding has advantages beyond that of compactness. Direct operations on BDDs are very powerful; they allow a compiler to use a high-level specification to describe complex queries and set transformations. Not only does this enable rapid development, but with a good ordering, one can expect good performance.

7. RELATED WORK

Points-to analysis [1, 9, 32] has been an active research field in the past several years. Hind [14] gives a very good overview of the current state of algorithms and metrics for points-to analysis. An important issue is how well the algorithms scale to large programs. Various points-to analyses make trade-offs between efficiency and precision. Equality-based analysis [32] runs in almost linear time, but with less precise results. On the other hand, subset-based analysis [1] produces more precise results, but with cubic worst-case complexity. In this work, we developed a specific version of a subset-based analysis that is suitable for implementing with BDDs, and which exhibits good space and time behaviour when used to analyze a range of Java programs, including a variety of large benchmark programs.

Various optimizations have been proposed to improve the efficiency of points-to analyses. Two of these optimizations, cycle elimination [10] and variable substitution [24], are based on the idea that variables whose points-to sets are provably equal can be merged, so that a single representation of the set can be shared

[3] If $|X|$ is size of the set, and M is the number bits used to encode the BDD, then the set can be enumerated in $\Theta(|X| \times M)$ time.

by multiple variables. Heintze and Tardieu [12] reported very fast analysis times using a demand-driven algorithm and a carefully designed implementation of points-to sets [11].

Several groups adapted the points-to analyses used for C to Java [19, 25]. These approaches, however, were applied only to benchmarks using the JDK 1.1.8 class library. One of the difficult points for whole program analysis for Java is that even very simple programs touch, or appear to touch, a large part of the class library. The JDK 1.3.1 class library is several times larger than the 1.1.8 library, and techniques which applied to the 1.1.8 case may no longer scale.

Recently, two approaches have been presented that have been shown to scale well to the JDK 1.3.1 library. Whaley and Lam [34] adapted the approach of Heintze and Tardieu to Java programs, and managed to get it to scale to benchmarks using the JDK 1.3.1 class library (although they made optimistic, potentially unsafe, assumptions about what part of the library needs to be analyzed). Lhoták and Hendren [17, 18] presented the SPARK framework, which allows experimentation with many variations of points-to analyses for Java. They used this framework to implement points-to solvers that were more efficient in time and space than the other reported work, including that of Whaley and Lam, making it the most efficient Java points-to analysis solver of which we are aware. Therefore, in our study of BDD-based points-to analysis, we used the SPARK framework both to generate the input to our BDD-based solver, and as a baseline solver against which to compare our new BDD-based solver.

Ordered Binary Decision Diagrams [5] represent boolean functions as DAGs. The canonical representation allows efficient boolean operations and testing of equivalence and satisfiability. Symbolic Model Checking [16] is used to verify circuits with an extremely large number of states by using BDDs. The use of BDDs in this context has allowed researchers to solve larger problems than can be solved using table-based representations of graphs. BDDs have also been used in software verification and program analyses. PAS [29] converts predicate and condition relations in a control flow graph to a compact BDD representation and performs analysis on these BDDs. Another use of BDDs is to represent large sets and maps; TVLA [21] and Bebop [3] are examples. In our work, as in model checking, we use BDDs to represent all data structures, and we show non-trivial techniques to make the original algorithm scalable to large programs using this new representation.

Although model checking and program analyses are not tightly connected yet, several publications have pointed out theoretical connections between them [26, 27]. The theoretical foundation of flow analyses is the fixed-point theory on monotonic functions, whose counterpart in model checking is the modal μ-calculus. Schmidt and Steffen [27] presented a methodology of treating iterative flow analysis as model checking of abstract interpretations. Bandera [8] is a tool-set applying such ideas to analyzing realistic programs. Like some other work [4], it abstracts program properties to linear temporal logic (LTL) or computational tree logic (CTL) formulas, which can be verified efficiently by existing model checking tools. Martena and Pietro [22] studied the application of a model checker, Spin, to solve intraprocedural alias analysis for C. In a different program analysis setting, BDD-based groundness analysis for constraint (logic) programs has become one of the standard approaches [13].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have 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 they 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.

It was not immediately obvious that a BDD-based approach would work for a program analysis like points-to. Although BDDs have been shown to be very effective in areas like hardware verification, program analyses face program properties that are quite different from those areas, including: 1) the problem may not be represented by LTL and CTL formulas; and 2) the analyzed object may not have many common patterns. For example, the transition system of a circuit written in CTL often exhibits regularities in the structure, which can be compactly represented in BDDs by applying good heuristics. However, before we started our work, it was not clear if the subset-inclusion relationship graph, and other data structures required for points-to analysis, had common structures that could give compactness. By systematically exploring a variety of orderings and empirically analyzing the performance, we did find an incrementalized algorithm and associated variable ordering that led to compact BDD representations. It is interesting to note that in the case of points-to analysis, it was not so important to find a compact representation for the **input** problem (unlike the case of hardware verification, where the input description may be very large and have many common patterns), but it was important to find a compact representation for the **solution** (i.e. the points-to relationships). Thus, it was the fact that the points-to sets showed a lot of regularity that leads to a fast and space-efficient solution. It would be very interesting to see if other whole-program analyses exhibit the same sort of regularity in their solutions. In our opinion, this is very likely.

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 [35] 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. Our preliminary experiments with using this technique on our points-to problem showed no significant improvement. In fact, the cost of reordering the large BDDs involved appeared to be even higher than the cost of solving the points-to problem with the original ordering. We therefore leave further investigation of dynamic variable reordering for BDD-based points-to analysis as future work.

In addition to achieving our goals in terms of time and space, we were pleasantly surprised with how easy it was for us to specify a wide variety of algorithms with the BDD approach. We tried many variations of the points-to analysis while developing our algorithm and it was very easy to go from one variation to the next. Based on this experience, we believe that a BDD package should be part of the standard toolkit for compiler analysis developers. Further, our BDD-based points-to analysis should be very easy to incorporate into program analysis tools where BDDs are used more and more frequently.

We plan to continue our work with BDDs and to further experiment with the kinds of queries outlined in Section 6. In addition, we would like to make a tighter connection between the Soot framework and a BDD toolkit so that subsequent BDD-based analyses could be specified at a very high level.

9. REFERENCES

[1] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language, May 1994. Ph.D thesis, DIKU, University of Copenhagen.

[2] Ashes suite collection `http://www.sable.mcgill.ca/software/`.

[3] Thomas Ball and Sriram K. Rajamani. Bebop: A Path-sensitive Interprocedural Dataflow Engine. In *Proceedings of PASTE'01*, pages 97 – 103, Jun 2001.

[4] David Basin, Stefan Friedrich, Marek Gawkowski, and Joachim Posegga. Bytecode Model Checking: An Experimental Analysis. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking Software, 9th International SPIN Workshop*, volume 2318 of *LNCS*, pages 42–59. Springer-Verlag, Apr 2002.

[5] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[6] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of PLDI'00*, pages 35–46, Jun 2000.

[7] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, Aug 1995. Springer.

[8] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, and W Visser. Tool-Supported Program Abstraction for Finite-State Verification. In *Proceedings of ICSE*, pages 177 – 187, 2001.

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of PLDI'94*, pages 242–256, 1994.

[10] Manuel Fähndrich, Jeffery S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of PLDI'98*, pages 85–96, Montreal, Canada, Jun 1998.

[11] Nevin Heintze. Analysis of large code bases: The compile-link-analyze model, 1999. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`.

[12] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of PLDI'01*, pages 254–263, June 2001.

[13] Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Evaluation of the domain Prop. *Journal of Logic Programming*, 23(3):237–278, 1995.

[14] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of PASTE'01*, pages 54 – 61, Jun 2001.

[15] jEdit: Open source programmer's text editor. `http://www.jedit.org/`.

[16] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401 – 424, 1994.

[17] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[18] Ondřej Lhoták 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.

[19] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proceedings of PASTE'01*, pages 73–79, 2001.

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

[21] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly Representing First-Order Structures for Static Analysis. In Manuel V. Hermenegildo and German Puebla, editors, *Proceedings of SAS'02*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer.

[22] Vincenzo Martena and Pierluigi San Pietro. Alias Analysis by Means of a Model Checker. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference*, volume 2027 of *LNCS*, pages 3–19, Genova, Italy, April 2001. Springer.

[23] Cristoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

[24] Atanas Rountev and Satish Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of PLDI'00*, pages 47 – 56, Jun 2000.

[25] Atanas Rountev, Ana Milanova, and Barbara Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of OOPSLA'01*, pages 43 – 55, 2001.

[26] David A. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretations. In *Proceedings of POPL'98*, pages 38 – 48, Jan 1998.

[27] David A. Schmidt and Bernhard Steffen. Program analysis *as* model checking of abstract interpretations. In *Proceedings of SAS'98*, pages 351 – 380, 1998.

[28] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of POPL'97*, pages 1–14, Paris, France, Jan 1997.

[29] John W. Sias, Wen mei W. Hwu, and David I. August. Accurate and Efficient Predicate Analysis with Binary Decision Diagrams. In *Proceedings of the 33rd annual IEEE/ACM International Symposium on Microarchitecture*, pages 112–123, Dec 2000.

[30] SPEC jbb200 benchmark `http://www.spec.org/osg/jbb2000/`.

[31] SPEC jvm98 benchmarks `http://www.spec.org/osg/jvm98/`.

[32] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of POPL'96*, pages 32–41, Jan 1996.

[33] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the 2000 OOPSLA*, pages 264–280, 2000.

[34] John Whaley and Monica Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of SAS'02*, volume 2477 of *LNCS*, 2002.

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

APPENDIX

A. FULL INCREMENTAL ALGORITHM

```
/* global variables, initialized by input constraints */
bdd pointsTo;   // points-to relation    V1 x H1
bdd edgeSet;    // assignment relation   V1 x V2
bdd stores;     // field store relation  V1 x (V2 x FD)
bdd loads;      // field load relation   (V1 x FD) x V2
bdd typeFilter; // encoding type filter  V1 x H1
/* caches for intermediate results */
bdd fieldPt;    // (H1 x FD) x H2 points-to relation for fields of heap objects
bdd storePt;    // H2 x (V1 x FD) temporary relation for field stores
bdd loadAss;    // (H1 x FD) x V2 temporary relation for field loads

/* incrementally computes points-to relation */
void solve_incremental(){
  bdd oldPointsTo = bdd_false();   // initialize variable to FALSE (0)
  bdd newPointsTo = pointsTo;
  // main iteratons
  do{
    // repeat rule (1) in the inner loop, see Figure 9 for details
    do{
      bdd newPt1 = bdd_relprod(edgeSet, newPointsTo, fdd_ithset(V1));
      bdd newPt2 = bdd_replace(newPt1, V2ToV1);
      bdd newPt3 = newPt2 - pointsTo;
      newPointsTo = newPt3 & typeFilter;
      pointsTo = pointsTo | newPointsTo;
    } while (newPointsTo != bdd_false());
    newPointsTo = pointsTo - oldPointsTo;

    // apply rule (2)
    bdd tmpRel1 = bdd_relprod(stores, newPointsTo, fdd_ithset(V1));     // (V2xFD)xH1
    bdd tmpRel2 = bdd_replace(bdd_replace(tmpRel1, V2ToV1), H1ToH2);    // (V1xFD)xH2
    bdd newStorePt = tmpRel2 - storePt;
    storePt |= newStorePt;                                              // (V1xFD)xH2

    bdd newFieldPt = bdd_relprod(storePt, newPointsTo, fdd_ithset(V1)); // (H1xFD)xH2
    newFieldPt |= bdd_relprod(newStorePt, oldPointsTo, fdd_ithset(V1)); // (H1xFD)xH2
    newFieldPt -= fieldPt;
    fieldPt |= newFieldPt;                                              // (H1xFD)xH2

    // apply rule (3)
    bdd tmpRel3 = bdd_relprod(loads, newPointsTo, fdd_ithset(V1));      // (H1xFD)xV2
    bdd newLoadAss = tmpRel3 - loadAss;
    bdd newLoadPt  = bdd_relprod(loadAss, newFieldPt, fdd_ithset(H1)&fdd_ithset(FD));
    newLoadPt |= bdd_relprod(newLoadAss, fieldPt, fdd_ithset(H1)&fdd_ithset(FD));
    loadAss |= newLoadAss;

    oldPointsTo = pointsTo;

    // convert new points-to relation to normal type
    newPointsTo = bdd_replace(bdd_replace(newLoadPt, V2ToV1), H2ToH1);
    newPointsTo -= pointsTo;

    // apply typeFilter
    newPointsTo = typeFilter & newPointsTo;
    pointsTo |= newPointsTo;

   // loop until points-to set has no changes
  } while (newPointsTo != bdd_false());
}
```