Using ZBDDs in Points-to Analysis

Ondřej Lhoták[1], Stephen Curial[2], and José Nelson Amaral[2]

[1] D. R. Cheriton School of Computer Science, University of Waterloo
[2] Department of Computing Science, University of Alberta

Abstract. Binary Decision Diagrams (BDDs) have recently become widely accepted as a space-efficient method of representing relations in points-to analyses. When BDDs are used to represent relations, each element of a domain is assigned a bit pattern to represent it, but not every bit pattern represents an element. The circuit design, model checking, and verification communities have achieved significant reductions in BDD sizes using Zero-Suppressed BDDs (ZBDDs) to avoid the overhead of these *don't-care* bit patterns. We adapt BDD-based program analyses to use ZBDDs instead of BDDs. Our experimental evaluation studies the space requirements of ZBDDs for both context-insensitive and context-sensitive program analyses and shows that ZBDDs can greatly reduce the space requirements for expensive context-sensitive points-to analysis. Using ZBDDs to reduce the size of the relations allows a compiler or other software analysis tools to analyze larger programs with greater precision. We also provide a metric that can be used to estimate whether ZBDDs will be more compact than BDDs for a given analysis.

1 Introduction

This paper describes improvements to Binary-Decision-Diagram-based implementations of pointer analysis used in ahead-of-time compilation and program analysis frameworks. The main benefit of BDDs [3] in program analysis is a reduction in the memory requirements of otherwise infeasible analyses: BDDs yield scalable highly context-sensitive may-point-to and call-graph-construction analyses [2, 12, 27, 30, 29]. The improvements presented in this paper further reduce the storage requirements, thus enabling more precise variations of the analysis to be computed for larger programs.

When various context-sensitive pointer analyses, such as that of Whaley and Lam [27] and object-sensitive analysis [16–18], were applied to object-oriented programs such as javac, soot, and the DaCapo benchmarks, the more precise variations (especially 3-object-sensitive, 1H-call-site sensitive, and Whaley/Lam) failed to complete due to memory limitations [13]. Although the running time of the most expensive analyses in [13] was several hours, none of the infeasible analyses failed to complete due to lack of time; they all failed due to excessive space requirements. A more compact BDD representation lowers the memory requirements of these analyses and allows them to scale to larger programs.

A BDD is a data structure representing a function that maps a vector of bits (the *BDD variables*) to a boolean value. When BDDs are used for program analysis, each element of the analysis is represented using some bit pattern. In general, however, not every bit pattern corresponds to an element, and these *don't-care* bit patterns unnecessarily increase BDD size.

A variation of BDDs, known as Zero-Suppressed BDDs (ZBDDs), are a promising alternative to eliminate the overhead of don't-care bit patterns [19, 23]. ZBDDs have been very effective at reducing BDD size in applications such as circuit design, model checking, and verification. Like these applications, program analyses use BDDs to represent and manipulate sets of elements chosen from a domain. Therefore, it is reasonable to expect these techniques to also reduce BDD size in program analysis. However, up to now there has been no description or evaluation of the use of ZBDDs in program analysis.

The main contributions of this paper are:

- A ZBDD representation of relations, and an algorithm, based on ZBDD multiplication, to compute the *relational product* on this representation. The combination of this representation and this algorithm makes it possible to use ZBDDs in relation-based program analysis.
- A ZBDD variation of the BDD-based points-to analysis of Berndl *et al.* [2].
- An empirical study of the space requirements of the ZBDD encodings of the relations in Berndl *et al.*'s analysis, Whaley and Lam's joeq/bddbddb [27], and Lhoták and Hendren's Paddle framework [10, 13].
- A relation-density metric that predicts whether a relation will be represented more compactly by a BDD or by a ZBDD.

The number of ZBDD variables used to represent a relation is the sum of the sizes of the domains of the relation. In analyses expressed using relational operations, the domains are sets of syntactic entities (such as statements, variables, etc.) of the program being analyzed, so this sum is linear in the size of the input. However, the context-sensitive call-graph specialization algorithm used in joeq/bddbddb [27] uses a special BDD operation with no relational equivalent to construct a single domain whose size is exponential in the size of the input. It is unlikely that an analogous operation can be practical for ZBDDs because it would have to construct a ZBDD over an exponential number of variables. Thus, we do not propose a ZBDD analogue of the joeq/bddbddb algorithm. However, other algorithms that use only relational operations and represent contexts as relations of syntactic entities can be implemented with our ZBDD representation. This includes the context-sensitive analyses in Paddle.

The rest of the paper is organized as follows. Section 2 gives background on pointer analysis, BDDs, and ZBDDs. Section 3 describes how ZBDDs can be used for program analysis. Section 4 compares the sizes of BDDs and ZBDDs used in pointer analysis. Section 5 reviews related work, and Section 6 concludes.

2 Background

A pointer analysis computes a static abstraction of the run-time relationships between pointers and their targets [6, 8]. For each static abstraction of a pointer variable, a points-to analysis computes a points-to set of the abstract target locations to which the variable points at run time. This work focuses on *may*-point-to information and on *subset-based* analysis. The result of a may-point-to analysis over-approximates the run-time relationships. In a *subset-based* analysis, also called Andersen-style analysis, points-to sets are computed by solving a collection of subset constraints [1]. Subset constraints are often solved by propagation. For example, the constraint $A \subseteq B$ can be satisfied by propagating the contents of A into B. Such propagation is done repeatedly for all the constraints in the system until a fixed-point solution satisfying all the constraints is reached.

A key difficulty is that the points-to sets can become very large, especially when precise abstractions of the run-time behavior are used. Recent research focus on efficient data structures and propagation algorithms. A BDD [3] is one such data structure [2, 29].

2.1 BDDs

A BDD represents a function that maps vectors of bits (the *BDD variables*) to boolean values. This function can be viewed as the set of bit vectors that the function maps to true. A BDD is a directed acyclic graph where a terminal node represents `true` and another terminal node represents `false`. Each non-terminal node, which specifies a BDD variable, has two outgoing edges to other nodes, a `one` edge and a `zero` edge. The value of the function for a given valuation of the BDD variables is determined by a traversal starting at the root node. At each node, the traversal follows either the one edge or the zero edge, depending on the value of the BDD variable associated with that node. The function has the value of the terminal node reached by the traversal.

f(x_1 x_2 x_3)	=
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	0

(a)

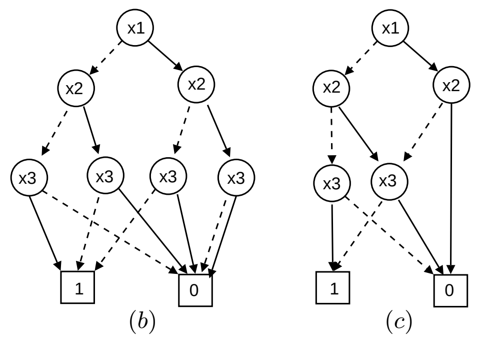

Fig. 1. The function f(x_1 x_2 x_3) (a), and the OBDD (b) and ROBDD (c) representing it. Solid edges represent 1-edges and dotted edges represent 0-edges.

An Ordered BDD (OBDD) is a BDD with a fixed variable ordering. Every path through an OBDD evaluates the variables in the given order. An OBDD can be reduced to remove redundant nodes, by following two reduction rules:

1. When two BDD nodes p and q are identical, edges leading to q are changed to lead to p, and q is eliminated from the BDD.
2. A BDD node p whose one-edge and zero-edge both lead to the same node q is eliminated from the BDD and the edges leading to p are redirected to q.

For any given function, the resulting Reduced Ordered BDD (ROBDD) is unique. Figure 1 shows an example function, an OBDD, and the ROBDD representing it. In practice, BDDs are always maintained in reduced ordered form. The remainder of this paper uses the abbreviation BDD to mean ROBDD.

In the BDD representation of a set, each element of each domain is encoded as a binary string. This encoding ideally uses the minimum number of bits required to assign each element to a unique binary string. A relation is formed by two or more attributes. Each attribute belongs to a domain and thus has a binary string representation. A relation can be represented as a set of binary strings by concatenating the binary encoding of each attribute. For example, assume a domain D with elements $\{a, b, c\}$ encoded as $\{00, 01, 10\}$, respectively, and a relation R that has 2 attributes $R_1 \in D$ and $R_2 \in D$. If R contains the tuples $\langle a, a\rangle$, $\langle a, b\rangle$ and $\langle c, b\rangle$, then R can be represented by the set $S = \{0000, 0001, 1001\}$. The BDD encoding of S evaluates to true for the strings in S and false for the strings not in S.

2.2 Solving Subset Constraints Using BDDs

Berndl et al. [2] and Zhu [29] show how to solve points-to subset constraints using BDDs. They encode both the points-to sets and subset constraints as relations represented with BDDs. Propagation is performed using the relational-product BDD operation. For example, consider a program with pointers p and q and abstract objects X and Y, with initial points-to sets $pt(p) = \{X\}$ and $pt(q) = \{Y\}$, and a subset constraint $pt(p) \subseteq pt(q)$. The relationships between pointers and abstract objects in this program are represented as a points-to relation $\{\langle X, p\rangle, \langle Y, q\rangle\}$ and a constraint relation $\{\langle p, q\rangle\}$. The result of propagating the original points-to sets along the constraint (which adds X to $pt(q)$) is computed by finding the relational product of the two relations (which evaluates to the relation $\{\langle X, q\rangle\}$).

2.3 ZBDDs

Zero-suppressed binary decision diagrams (ZBDDs) are like BDDs (see Section 2.1), but the second reduction rule is changed to:

2. A BDD node p whose one-edge leads to the zero terminal node and whose zero-edge leads to a node q is removed from the BDD and the edges leading to p are redirected to q.

Because of the difference in the reduction rules, the interpretation of a ZBDD is slightly different than a BDD. To determine the value of the function for a given valuation of the ZBDD variables, the ZBDD is traversed like a BDD, following either the one or zero edge of each node depending on the value of the variable tested by the corresponding node. However, the final value is true only if the traversal ends at the `true` terminal node *and every variable whose value is 1 has been tested during the traversal.* Otherwise, the final value is false. For example, the function that was presented in Figure 1(a) is represented by the ZBDD in Figure 2. For instance, according to the ZBDD interpretation, the bit pattern `011` maps to `false` because the `true` terminal is reached without testing variable `x3`. Because of the difference in reduction rules, ZBDDs can represent some functions more compactly than BDDs, and vice versa.

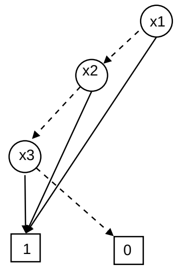

Fig. 2. ZBDD representation of the function from Figure 1.

3 Encoding relations in ZBDDs

In a one-of-N encoding, the number of bits used is equal to the size of the domain. Each element is associated with one bit in the vector. Each element is represented by a bit vector with 1 for the corresponding element and 0 elsewhere. ZBDDs are particularly suited to manipulate sets encoded using a one-of-N encoding. According to Meinel and Theobald [15, p. 224], ZBDDs compactly encode sets of bit vectors that are sparse in the sense that: (i) the set contains only a small number of bit vectors relative to the number of all possible combinations of n bits; and (ii) each bit vector in the set contains few one bits.

The first condition holds in many practical problems involving sets. The second condition is a consequence of the one-of-N encoding. ZBDDs are more efficient than BDDs in many set-based applications, such as combinatorial problems [20], problems in graph theory [4], and traversal of Petri nets [28]. Since points-to analysis also requires manipulation of (points-to) sets, ZBDDs should also work well for points-to analysis.

Although one-of-N encodings implemented using ZBDDs have been used successfully in problems involving sets, relatively little attention has been paid to encoding relations. A relation is a subset of a cross product of its attributes. The size of this universal set is the product of the sizes of the attribute domains, which can be very large. Encoding a relation in a ZBDD as a subset of this universal set is not practical because the number of bits required is equal to the size of the universal set. Yoneda *et al.* come close to manipulating relations in ZBDDs [28]. Although they do not represent relations explicitly, they define new ZBDD operations that have the effect of applying a transition relation to a set of Petri-net states encoded in a ZBDD.

We propose a new technique to represent a relation in a ZBDD: allocate one bit for each element of every attribute domain. Thus, the number of bits required is the sum, rather than the product, of the sizes of the attribute domains. A tuple containing one element from each attribute is represented as a set of those elements. For example, suppose a domain D with elements $\{a, b, c\}$, and a relation R with two attributes R_1 and R_2 with domain D. Encode this relation as a ZBDD on six bits, namely $a_1, b_1, c_1, a_2, b_2, c_2$, where the bits with subscript 1 represent elements in attribute 1, and the bits with subscript 2 represent elements in attribute 2. Then the tuples $\langle a, a\rangle$, $\langle a, b\rangle$ and $\langle c, b\rangle$ are represented by the sets $\{a_1, a_2\}$, $\{a_1, b_2\}$, and $\{c_1, b_2\}$, and encoded with the binary strings 100100, 100010, and 001010, respectively.

This representation encodes each tuple as a bit vector. Therefore the standard ZBDD set operations defined on sets of bit vectors (union, intersection, difference) implement the corresponding operations on the relations. The replace operation can be implemented on ZBDDs in the same way as on BDDs.

The relational product operation is central to relation-based points-to analysis. To our knowledge, there is no practical algorithm to compute the relational product in ZBDDs. In BDDs a relational product is a conjunction followed by an existential quantification — implementations combine them into a single, more efficient, operation. In ZBDDs, the analogue of the conjunction is a multiplication followed by removal of tuples containing more than one element of the attribute being compared.

For instance, the example from Section 2.2. has points-to relation $\{\langle p, X\rangle, \langle q, Y\rangle\}$ and subset constraints relation $\{\langle p, q\rangle\}$. These relations can be represented using ZBDDs with bits X, Y, p, q, p', q', where the primed bits represent elements in the second attribute of the relation. The points-to relation is represented by a ZBDD for the set of subsets $\{Xp, Yq\}$, and the subset constraints relation is represented by $\{pq'\}$. The product of these ZBDDs is $\{Xpq', Yqpq'\}$. The second tuple is removed because it contains two elements (p and q) from the attribute being compared. Finally, the equivalent of an existential quantification removes the p from Xpq', yielding the correct final result $\{Xq'\}$.

Although an algorithm for ZBDD multiplication is given by Minato [21, p. 75], there are two other operations for which algorithms have not been designed: (1) removal of tuples with multiple elements from the same attribute; and (2) existential quantification. We present a modification of the ZBDD multiplication algorithm (see Figure 3) that performs all three operations in a single pass through the ZBDD. ZRelProd takes an additional parameter *pd*, the set of ZBDD variables representing the relation attributes being compared. Let x be the variable tested by the top node of the operand ZBDDs. When x is not in *pd*, line 14 performs the standard multiplication.[3] However, when x is in *pd*, line 12 returns the union of two relational products: the product of the 0-cofactors with respect to x and the product of the 1-cofactors with respect to x. This result

[3] Compared to Minato's ZBDD multiplication algorithm, line 14 lacks the terms $p1*q1$ and $p0*q1$. This is a relational-product behaviour-preserving optimization: when x is not in *pd* but it is tested by p, it cannot also be tested by q, so $q0 = q$ and $q1 = 0$.

```
ZBDD ZRelProd(ZBDD $p$, ZBDD $q$, Set⟨Variable⟩ $pd$)
if $p.top < q.top$
  then return ZRelProd$(q, p, pd)$
if $q = 0$
  then return $0$
if $q = 1$
  then return Subset0$(p, pd)$
$x \leftarrow p.top$
$(p0, p1) \leftarrow$ factors of $p$ by $x$
if $x \in pd$
  then
      $(q0, q1) \leftarrow$ factors of $q$ by $x$
      return ZRelProd$(p1, q1, pd)$ + ZRelProd$(p0, q0, pd)$
  else
      return $x \cdot$ ZRelProd$(p1, q, pd)$ + ZRelProd$(p0, q, pd)$
```

Fig. 3. The Relational Product Algorithm for ZBDDs.

contains exactly those tuples in which the value of x is equal in both operands. Tuples in which the value of x is zero appear in the 0-cofactors, and those in which x is one appear in the 1-cofactors. ZRelProd combines into a single step the ZBDD multiplication, the removal of tuples with multiple elements from the same attribute, and the computation of the existential quantification. The following theorem shows its correctness.

Theorem 1. *Let $V = \{v_1 \ldots v_n\}$ be a set of ZBDD variables, ordered such that if a ZBDD node testing v_i is a child of a ZBDD node testing v_j, then $i < j$ (i.e. v_1 is closest to the terminal nodes). Partition V into three disjoint subsets V_1, V_2, V_3 representing the domains unique to the left-hand-side relation, the domains common to both relations, and the domains unique to the right-hand-side relation. Let $P \subseteq \mathcal{P}(V_1 \cup V_2)$ and $Q \subseteq \mathcal{P}(V_2 \cup V_3)$ be arbitrary sets of subsets of $V_1 \cup V_2$ and $V_2 \cup V_3$ represented as ZBDDs. Define*

$$P \times Q = \{s_1 \cup s_3 : \exists s_2 \subseteq V_2 . s_1 \cup s_2 \in P \wedge s_2 \cup s_3 \in Q \wedge (s_1 \cup s_3) \cap V_2 = \emptyset\}$$

Then ZRelProd$(P, Q, V_2) = P \times Q$*. That is,* ZRelProd *correctly computes the relational product of the relations represented by P and Q.*

Proof. Define $k(P) = \max\{i : v_i \in S \wedge S \in P\}$, with $k(P) = 0$ when P is the empty set or contains only the empty set. Then the top (root) node of the ZBDD representing P tests variable $v_{k(P)}$, since a node that tests $v_{k(P)}$ must appear in the ZBDD in order for $v_{k(P)}$ to appear in a set in P, and the maximality of $k(P)$ ensures that this node is at the top of the ZBDD. Define operations $s0(P, v_i) = \{S : S \in P \wedge v_i \notin S\}$ and $s1(P, v_i) = \{S \setminus \{v_i\} : S \in P \wedge v_i \in S\}$, which partition P into those sets that do not contain v_i and those that do, and remove v_i from each set in the latter partition. The cofactor ZBDD operation computes $s0$ and $s1$.

The proof is by induction on $K = \max\{k(P), k(Q)\}$. In the base case, $k(P) = k(Q) = 0$, so Q is either the empty set or contains only the empty set. When $Q = \emptyset$, $P \times Q = \emptyset$, and line 4 correctly returns the ZBDD representing the empty set. When Q is the set containing the empty set, $P \times Q$ is the set of sets from P not containing any elements of V_2. The SUBSET0 ZBDD operation computes this set in Line 6.

In the inductive case, if $k(P) < k(Q)$, the algorithm switches P and Q; since $\times$ is symmetric, we need only consider the case when $k(P) \geq k(Q)$, so $K = k(P)$. When $k(Q) = 0$, the same argument as for the base case applies. Thus, consider the case when $k(Q) > 0$, so line 7 of the algorithm is reached. There are two cases to consider: either $v_{k(P)} \in V_1$ or $v_{k(P)} \in V_2$.

Case 1: $v_{k(P)} \in V_1$: Partition $P \times Q$ into $R1 = \{s \in P \times Q : v_{k(P)} \in s\}$ and $R0 = \{s \in P \times Q : v_{k(P)} \notin s\}$. Define $v_{k(P)} \cdot X = \{S \cup \{v_{k(P)}\} : S \in X\}$. From the definition of $\times$, $s0(P, v_{k(P)}) \times Q = R0$, and $v_{k(P)} \cdot (s1(P, v_{k(P)}) \times Q) = R1$. Since $v_{k(P)} \notin V_2$, the condition in line 9 fails and line 14 is executed. By the definition of the cofactor operation, neither $p0$ nor $p1$ contains any sets containing $v_{k(P)}$, so $k(p0) < k(P) = K$ and $k(p1) < k(P) = K$. Since no set in Q contains an element of V_1, $k(Q) < k(P) = K$. Thus, the inductive hypothesis can be applied to the relational products in line 14 to show that they compute $s1(P, v_{k(P)}) \times Q$ and $s0(P, v_{k(P)}) \times Q$, respectively. Adding $v_{k(P)}$ (i.e. x) to each set in the former and taking their union, as done in line 14, gives $R1 \cup R0 = P \times Q$ as required.

Case 2: $v_{k(P)} \in V_2$: In the definition of $\times$, for each element of $P \times Q$, there must exist some s_2. Partition $P \times Q$ into $R1$ containing those elements for which $v_{k(P)} \in s_2$, and $R0$ containing those elements for which $v_{k(P)} \notin s_2$. From the definition of $\times$, $s1(P, v_{k(P)}) \times s1(Q, v_{k(P)}) = R1$ and $s0(P, v_{k(P)}) \times s0(Q, v_{k(P)}) = R0$. Since $v_{k(P)} \in V_2$, the condition in line 9 succeeds and lines 11 and 12 are executed. Again, by the definition of the cofactor operation, $k(p0), k(p1), k(q0)$, and $k(q1)$ are all strictly less than $k(P) = K$, so the inductive hypothesis can be applied to show that the relational products in line 12 correctly compute $s1(P, v_{k(P)}) \times s1(Q, v_{k(P)})$ and $s0(P, v_{k(P)}) \times s0(Q, v_{k(P)})$. Line 12 returns their union, which is $R1 \cup R0 = P \times Q$ as required. $\square$

One other issue with ZBDDs is that the set-complement operation cannot be performed efficiently because the complement of a sparse set is no longer sparse. The BDD-based points-to analyses of Berndl *et al.* [2] and of Whaley and Lam [27] do not use set complement. The Paddle framework [10] uses set complement for convenience (in cases where it is more natural to write $R_1 \cap \overline{R_2}$ instead of $R_1 \setminus R_2$) but not in essential ways. Paddle could be restructured to avoid using set complement.

4 Experimental Evaluation

The program analysis community started using BDDs to represent relations without investigating whether a variant representation could be more compact. The experiments presented in this section test whether ZBDDs are a better choice of data structure for program analyses.

The results indicate that ZBDDs are consistently more space efficient than BDDs for relations in context-sensitive points-to analyses, but yield little improvement for the dense relations found in context-insensitive points-to analyses.

4.1 Experimental Setup

This experimental study evaluates ZBDDs in the context of three program-analysis frameworks.

- The first framework is the context-insensitive points-to analysis developed by Berndl *et al.* [2]. In this implementation, Soot [26] and its Spark points-to analysis framework [9, 11] are used to generate a system of subset constraints to be solved. The constraints are then read in and solved by a solver written in C using the BuDDy BDD library [14].
- The second framework is the joeq/bddbddb system of Whaley and Lam [27]. In this implementation, the joeq compiler pre-processes the code to be analyzed, generates a system of subset constraints to be solved, and outputs the initial relations as BDDs. The algorithm to solve the constraints is specified as a Datalog program. The bddbddb tool reads the Datalog program and the initial relations, and solves the system of constraints. We evaluated ZBDDs within the context-insensitive points-to analysis implemented in joeq/bddbddb. As explained in the introduction, we did not apply ZBDDs to the context-sensitive analysis in joeq/bddbddb because it uses a "new primitive" BDD operation to construct domains of exponential size [27].
- The third framework is Lhoták and Hendren's Paddle framework [10, 13]. Unlike the other two systems, Paddle integrates the BDD-based analysis into the compiler (Soot). Paddle is implemented in the Jedd language [12], an extension of Java for expressing program analyses in terms of relations, which the Jedd runtime represents and manipulates using BDDs. All modifications are confined to Jedd. Of the variations of context sensitivity supported by Paddle, we evaluated the 1-object-sensitive analysis, which was identified in earlier work as being precise at a modest cost, relative to other context sensitivity variations [13].

This study uses a representative subset of the benchmarks from Lhoták and Hendren's study [13] of context-sensitive points-to analysis. Three (`antlr`, `bloat`, `chart`) are from the Dacapo suite, version beta050224 [5], four (`jack`, `javac`, `jess`, `raytrace`) are object-oriented programs from the SPEC JVM 98 suite [25], and three (`polyglot`, `sablecc`, `soot`) are other object-oriented Java programs. These benchmarks have been used in many previous points-to analysis studies. All of the benchmarks are analyzed with the standard class library from the Sun JDK 1.3.1.

Operations on BDDs and ZBDDs have the same asymptotic complexity. Packages such as BuDDy contain tuned implementations of BDD operations. We did not perform a comparison of running time because we do not have access to a carefully-tuned ZBDD implementation. Unlike running time, the number of nodes is not affected by the fine tunning of the decision diagram implementation.

4.2 ZBDDs

This study compares ZBDDs to BDDs in two ways. First, we wrote a variation of the BDD-based points-to analysis implementation of Berndl *et al.* [2] that represents the same relations in ZBDDs instead of BDDs. This variation uses ZBDD operations, including the relational-product operation presented in Section 3, instead of BDD operations to manipulate relations. Second, we instrumented Paddle [10, 13] and joeq/bddbddb [27] to dump the relations computed during the analysis. Then we developed a tool to read these relations into both a BDD and a ZBDD. This infrastructure allows for the comparison of the number of nodes in each BDD with the number of nodes in the corresponding ZBDD representing the same relation.

A fair comparison must use an appropriate ordering of the variables in the BDD and ZBDD because the size of these representations can vary significantly depending on the choice of ordering. Berndl *et al.* found that requiring that all the bits representing a given attribute be grouped together consecutively in the ordering is a suitable restriction for BDD-based program analyses [2]. Thus, for each relation, we searched exhaustively for the BDD ordering that obeys this restriction and produces the smallest BDD: for a relation with n attributes, we evaluated the $n!$ possible orderings of the attributes. The ordering found by this exhaustive search on a representative benchmark (`antlr`) was applied to the corresponding relations in the analysis of all the benchmarks.

Since the BDD and ZBDD representations of a relation are defined in terms of different sets of bits, an appropriate ZBDD variable ordering that corresponds to a given BDD variable ordering must be selected. To be consistent with BDD orderings and to limit the search space of possible orderings, the restriction that the bits representing a given attribute be grouped together consecutively is also maintained for ZBDD orderings. Given this restriction, the only choice remaining is the relative ordering of different attributes. It turns out that for most of the relations examined, the best BDD variable ordering is also the best ZBDD ordering. For all but one relation, using the best BDD ordering for the ZBDD results in a ZBDD no more than 5% larger than the ZBDD with the best ZBDD ordering.

The `resolvedSpecials` relation in Paddle is an interesting outlier. Using the best BDD ordering results in a ZBDD 76% larger than the ZBDD with the best ZBDD ordering. However, when the best ZBDD ordering is applied to BDDs it yields only 7% more nodes than the best BDD ordering. We will continue to study such outliers for more insights on BDD and ZBDD orderings. However, for the most part, good BDD orderings tend to also be good ZBDD orderings. The experiments reported in the remainder of this paper use the best BDD ordering for both BDDs and ZBDDs. Therefore, the results are slightly biased in favour of BDDs.

The graphs in Figure 4 show the relative size of the BDD and ZBDD for each relation of each benchmark. Points below the diagonal line represent relations for which the ZBDD is smaller than the BDD. Points above the line represent relations for which the BDD is smaller. Larger decision diagrams, which affect

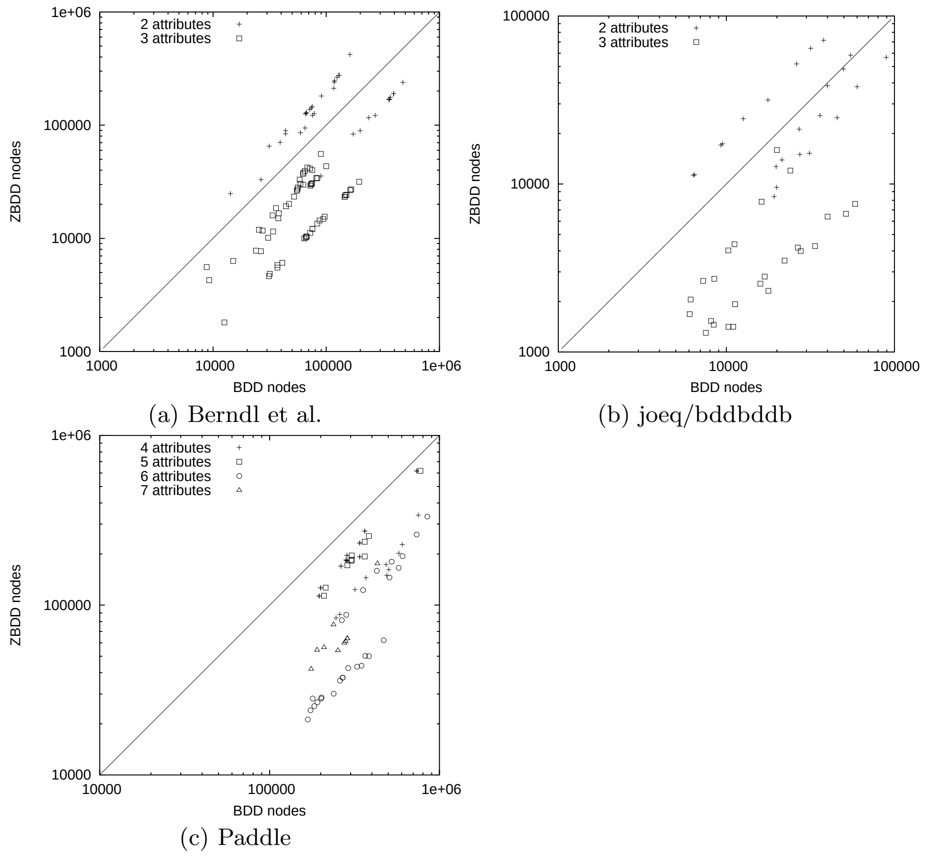

Fig. 4. BDD size compared to ZBDD size

analysis cost more significantly, appear to the right and top of each graph. Note that the three graphs in Figure 4 have different scales.

In the Berndl *et al.* analysis, although most relations are represented more efficiently by ZBDDs than BDDs, the reverse is true for a significant number of relations, some of them large. Closer examination reveals that in every benchmark, the relations represented more efficiently by BDDs than ZBDDs are always the `pointsTo` and `typeFilter` relation, both of which are manipulated frequently by the analysis. Thus, the data indicates that there is no clear advantage in using ZBDDs over BDDs, or vice versa, for the Berndl *et al.* analysis.

Results for the joeq/bddbddb analysis are similar to those of the Berndl *et al.* analysis. The relations for which BDDs are smaller than ZBDDs are mainly `vPfilter` (the joeq/bddbddb equivalent of `typeFilter`), and in several benchmarks `vP` (equivalent of `pointsTo`).

In both the Berndl *et al.* and joeq/bddbddb analyses, the relative size of BDDs and ZBDDs favours ZBDDs more strongly in relations with three attributes than those with two attributes. Therefore, for context-sensitive analyses, which use additional attributes to represent contexts, we expected ZBDDs to be significantly smaller than BDDs. Indeed, Figure 4(c) shows that in the Paddle context-sensitive analysis every relation in every benchmark is smaller when represented by a ZBDD than by a BDD. The differences ranged up to a factor of eight! However, the link with the number of attributes is less clear in the Paddle results: (1) the BDD and ZBDD sizes for 6-attribute relations are very close to those for 4-attribute relations; and (2) the ZBDD vs. BDD advantage is smaller for 7-attribute relations than for a large set of 6-attribute relations.

These results indicate that for context-insensitive points-to analyses, ZBDDs are generally smaller than BDDs, but the advantage is too small and inconsistent to allow a general recommendation that ZBDDs be used instead of BDDs. However, for analyses using relations with more attributes, and for context-sensitive points-to analysis in particular, we expect ZBDDs to be significantly and consistently smaller than BDDs representing the same relations.

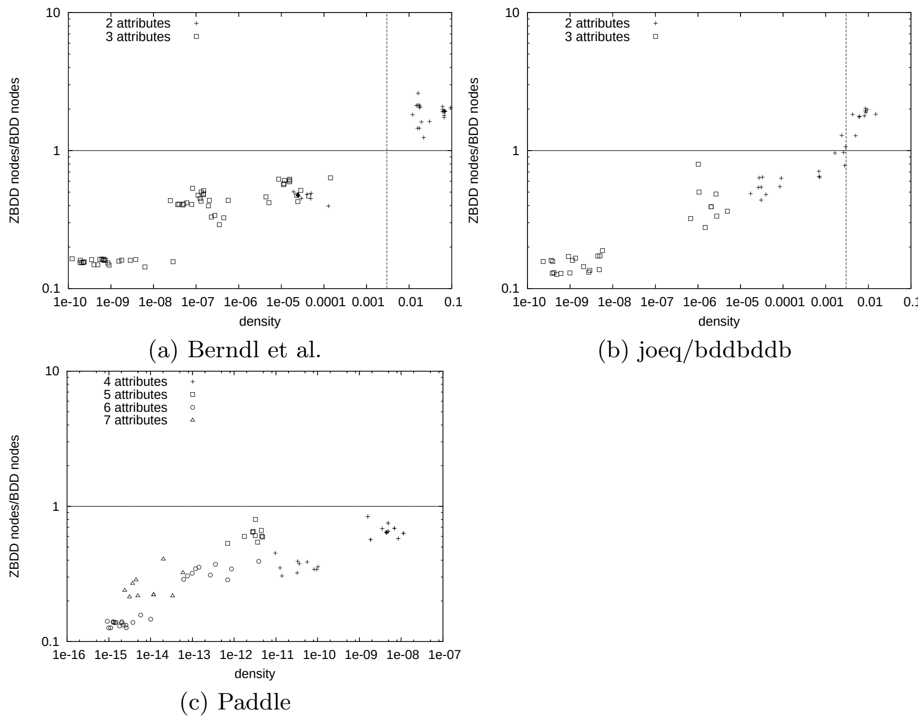

Fig. 5. BDD vs. ZBDD size in terms of density

When should we use ZBDDs? Given the mixed results for points-to analyses in the question of the size of ZBDDs vs. BDDs for the same relations, a metric that indicates whether ZBDDs or BDDs are expected to be more compact would be useful for the program analysis community. Such a metric could be used when a designer is considering the use of a BDD representation for a relation-based analysis that has not yet been implemented using BDDs.

The metric that we propose is *density* and it is equal to the number of tuples in the relation divided by the size of the full domain of possible tuples. This metric is inspired by Meinel and Theobald's recommendation that ZBDDs be used for boolean functions whose on-set (those input vectors that the function maps to one) is small, and whose bit vectors in the on-set contain few one-bits [15]. The density metric covers both parts of this qualitative recommendation, since the number of tuples in a relation is equal to the total number of one-bits in all the bit vectors in the on-set of a one-of-N encoding of the relation, multiplied by the number of attributes. The density metric measures the density of a relation. Thus it applies to relations independently of the ZBDD representation.

Figure 5 plots the ratio of ZBDD vs. BDD size as a function of the density metric. Points below the horizontal line represent relations whose ZBDD is smaller than the BDD. In all three graphs, as density increases, the advantage of ZBDDs over BDDs decreases. At a density of around 3×10^{-3}, BDDs and ZBDDs are approximately equal in size. This threshold is indicated in the graphs by a vertical dotted line. Of all the relations that we observed, two had a density lower than this threshold but were represented more compactly by a BDD than a ZBDD; they appear slightly to the left of and above the crossing lines in Figure 5(b). The context-sensitive relations extracted from Paddle, which are represented more compactly by ZBDDs than BDDs, have low densities. Because the size of BDDs and ZBDDs strongly depends on the contents of the relation being represented, density can only serve as a rough guide. However, we hope it will be a useful metric for analysis designers considering ZBDDs or BDDs for other program analyses.

5 Related Work

ZBDDs were introduced by Minato, and found to scale better than BDDs when representing large combinatorial circuits [19]. Minato showed that ZBDDs are likely a better choice than BDDs if there are many input variables, variables default to 0, or very few elements in a set are asserted.

Okuno applied ZBDDs to the N-Queens problem. He reports that for this problem, the ZBDD representation is about a factor of N smaller than the corresponding BDD version [24, summarized in [21]].

Yoneda *et al.* applied ZBDDs to Petri-net state-space exploration, and compared their performance to BDDs [28]. They found the ZBDD representation to be one half to one third the size of the BDD representation. They also report that the ZBDD implementation was several times faster for some benchmarks.

Coudert used ZBDDs to efficiently solve graph optimization and routing problems [4].

Since then, ZBDDs have been used to efficiently solve several combinatorial problems as well as fault simulation, logic synthesis, processing of petri nets and manipulation of polynomial formulas [22, 23].

BDDs were first applied to points-to analysis by Zhu and Berndl *et al.* [29, 2]. These context-insensitive analyses were then generalized to context-sensitive analysis by Zhu and Calman and by Whaley and Lam [30, 27]. Hardekopf and Lin compared several non-BDD and BDD implementations of a context-insensitive points-to analysis [7], including a hybrid implementation in which only the points-to sets are represented by BDDs, and the rest of the analysis uses traditional data structures. This implementation used less than one-fifth of the memory of a non-BDD implementation. ZBDDs could be substituted for BDDs in this implementation, possibly yielding further reductions in memory usage.

6 Conclusion

Although BDDs have been successfully used for points-to analysis, alternative BDD representations were not evaluated by this community. This paper develops the techniques that allow the use of ZBDDs for such analyses. The new relational-product operator described here allows for the immediate use of ZBDDs in points-to analysis. The experimental results indicate that non-trivial reduction of BDD sizes can be realized when ZBDDs are used for context-sensitive points-to analysis.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).
2. M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of PLDI 2003*, pages 103–114, 2003.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
4. O. Coudert. Solving graph optimization problems with ZBDDs. In *EDTC '97: Proceedings of the 1997 European Conference on Design and Test*, page 224, 1997.
5. DaCapo Project. The DaCapo benchmark suite. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.
6. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of PLDI 1994*, pages 242–256, 1994.
7. B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of PLDI 2007*, 2007.
8. M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of PASTE 2001*, pages 54–61. ACM Press, 2001.
9. O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, Dec. 2002.
10. O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.

11. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Compiler Construction, 12th International Conference*, pages 153–169. Springer, 2003.
12. O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of PLDI 2004*, pages 158–169. ACM Press, 2004.
13. O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction, 15th Int. Conf.*, pages 47–64. Springer, 2006.
14. J. Lind-Nielsen. BuDDy, A Binary Decision Diagram Package. `http://www.itu.dk/research/buddy/`.
15. C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design.* Springer-Verlag New York, Inc., 1998.
16. A. Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software.* PhD thesis, Rutgers University, Aug. 2003.
17. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of ISSTA 2002*, pages 1–11. ACM Press, 2002.
18. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.
19. S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC '93: 30th International Conf. on Design Automation*, pages 272–277, 1993.
20. S. Minato. Calculation of unate cube set algebra using zero-suppressed BDDs. In *31st ACM/IEEE Design Automation Conference (DAC '94)*, pages 420–424, 1994.
21. S. Minato. *Binary decision diagrams and applications for VLSI CAD.* Kluwer Academic Publishers, 1996.
22. S. Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):156–170, May 2001.
23. A. Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, Portland State University, June 2001.
24. H. G. Okuno. Reducing combinatorial explosions in solving search-type combinatorial problems with binary decision diagrams. *Trans. of Information Processing Society of Japan (IPSJ), (in Japanese)*, 35(5):739–753, May 1994.
25. Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.
26. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
27. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of PLDI 2004*, pages 131–144, 2004.
28. T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. zero-suppressed BDDs : for CTL symbolic model checking of petri nets. In *Formal Methods in Computer-Aided Design*, pages 435–449, 1996.
29. J. Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 150–157, 2002.
30. J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of PLDI 2004*, pages 145–157, 2004.