Using Datalog with Binary Decision Diagrams for Program Analysis

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam

Computer Science Department
Stanford University
Stanford, CA 94305, USA
{jwhaley, dzin, mcarbin, lam}@cs.stanford.edu

Abstract. Many problems in program analysis can be expressed naturally and concisely in a declarative language like Datalog. This makes it easy to specify new analyses or extend or compose existing analyses. However, previous implementations of declarative languages perform poorly compared with traditional implementations. This paper describes bddbddb, a BDD-Based Deductive DataBase, which implements the declarative language Datalog with stratified negation, totally-ordered finite domains and comparison operators. bddbddb uses binary decision diagrams (BDDs) to efficiently represent large relations. BDD operations take time proportional to the size of the data structure, not the number of tuples in a relation, which leads to fast execution times. bddbddb is an effective tool for implementing a large class of program analyses. We show that a context-insensitive points-to analysis implemented with bddbddb is about twice as fast as a carefully hand-tuned version. The use of BDDs also allows us to solve heretofore unsolved problems, like context-sensitive pointer analysis for large programs.

1 Introduction

Many program analyses can be expressed naturally and easily in logic programming languages, such as Prolog and Datalog [14, 29, 36]. Expressing a program analysis declaratively in a logic programming language has a number of advantages. First, analysis implementation is greatly simplified. Analyses expressed in a few lines of Datalog can take hundreds to thousands of lines of code in a traditional language. By automatically deriving the implementation from a Datalog specification, we introduce fewer errors. Second, because all analysis information is expressed in a uniform manner, it is easy to use analysis results or to combine analyses. Finally, optimizations of Datalog can be applied to all analyses expressed in the language.

However, implementations using logic programming systems are often slower than traditional implementations and can have difficulty scaling to large programs. Reps reported an experiment using Corel, a general-purpose logic programming system, to implement on-demand interprocedural reaching definitions analysis [29]. It was found that the logic programming approach was six times slower than a native C implementation. Dawson et al. used Prolog to perform groundness analysis on logic programs and strictness analysis on functional programs [14]. Using the XSB system, which has better efficiency than Corel [31], they were able to analyze a number of programs efficiently. However, the programs they analyzed were small — under 600 lines of code.

Other recent work by Liu and Stoller on efficient Datalog appears to have promise, but they do not present any performance results [21].

Our system for specifying program analyses using Datalog has successfully analyzed programs with tens of thousands of lines of source code, and has regularly performed faster than handcoded analyses. We will discuss our experience developing points-to analyses for C and for Java, which we compare with earlier handcoded versions. We will also discuss the results of using a security analysis and an external lock analysis which were specified using the same system.

1.1 Datalog using BDDs

We have developed a system called bddbddb, which stands for BDD-Based Deductive DataBase. bddbddb is a solver for Datalog with stratified negation, totally-ordered finite domains and comparison operators. bddbddb represents relations using binary decision diagrams, or BDDs. BDDs are a novel data structure that were traditionally used for hardware verification and model checking, but have since spread to other areas. The original paper on BDDs is one of the most cited papers in computer science [8].

Recently, Berndl et al. showed that BDDs can be used to implement context-insensitive inclusion-based pointer analysis efficiently [5]. This work showed that a BDD-based implementation could be competitive in performance with traditional implementations. Zhu also investigated using BDDs for pointer analysis [40, 41]. In 2004, Whaley and Lam showed that BDDs could actually be used to solve context-sensitive pointer analysis for large programs with an exponential number of calling contexts, a heretofore unsolved problem [38]. Thus, by using BDDs, we can solve new, harder program analysis problems for which there are no other known efficient algorithms.

Datalog is a logic programming language designed for relational databases. We translate each Datalog rule into a series of BDD operations, and then find the fixpoint solution by applying the operations for each rule until the program converges on a final set of relations. By using BDDs to represent relations, we can use BDD operations to operate on entire relations at once, instead of iterating over individual tuples.

Our goal with bddbddb was to hide most of the complexity of BDDs from the user. We have several years of experience in developing BDD-based program analyses and we have encoded our knowledge and experience in the design of the tool. Non-experts can develop their own analyses without having to deal with the complexities of fine-tuning a BDD implementation. They can also easily extend and build on top of the results of advanced program analyses that have been written for bddbddb.

Using bddbddb is not only easier than implementing an analysis by hand — it can also produce a more efficient implementation. bddbddb takes advantage of optimization opportunities that are too difficult or tedious to do by hand. We implemented Whaley and Lam's context-sensitive pointer analysis [38] using an earlier version of the bddbddb system and found that it performed significantly faster than a hand-coded, hand-tuned implementation based on BDDs. The hand-coded implementation, which was 100 times longer, also contained many more bugs.

We and others have also used bddbddb for a variety of other analyses and analysis queries, such as C pointer analysis, eliminating bounds check operations [1], finding security vulnerabilities in web applications [22], finding race conditions, escape analysis, lock analysis, serialization errors, and identifying memory leaks and lapsed listeners [23].

1.2 Contributions

This paper makes the following contributions:

1. Description of the bddbddb system. This paper describes in detail how bddbddb translates Datalog into efficient, optimized BDD operations and reports on the performance gains due to various optimizations. We expand upon material introduced in an earlier tutorial paper [18].
2. Demonstration of effective application of logic programming to problems in program analysis. Whereas previous work shows that there is a penalty in writing program analysis as database operations, we show that a BDD implementation of Datalog for program analysis can be very efficient. Interprocedural program analysis tends to create data that exhibits many commonalities. These commonalities result in an extremely efficient BDD representation. Datalog's evaluation semantics directly and efficiently map to BDD set operations.
3. Experimental results on a variety of program analyses over multiple input programs show that bddbddb is effective in generating BDD analyses from Datalog specifications. In particular, we compare bddbddb to some hand-coded, hand-optimized BDD program analyses and show that bddbddb is twice as fast in some cases, while also being far easier to write and debug.
4. Insights into using BDDs for program analysis. Before building this tool, we had amassed considerable experience in developing BDD-based program analyses. Much of that knowledge went into the design of the tool and our algorithms. This paper shares many of those insights, which is interesting to anyone who uses BDDs for program analysis.

1.3 Paper Organization

The rest of the paper is organized as follows. We first describe how a program analysis can be described as a Datalog program in Section 2. Section 3 deconstructs a Datalog program into operations in relational algebra, and shows how BDDs can be used to represent relations and implement relational operations. Section 4 describes the algorithm used by bddbddb to translate a Datalog program into an interpretable program of efficient BDD operations. Section 5 presents experimental results comparing bddbddb to hand-coded implementations of program analysis using BDDs. In Section 6 we discuss the related work. Our conclusions are in Section 7.

2 Expressing a Program Analysis in Datalog

Many program analyses, including type inference and points-to analyses, are often described formally in the compiler literature as inference rules, which naturally map to Datalog programs. A program analysis expressed in Datalog accepts an input program represented as a set of input relations and generates new output relations representing the results of the analysis.

2.1 Terminology

bddbddb is an implementation of Datalog with stratified negation, totally-ordered finite domains, and comparison operators. A Datalog program P consists of a set of domains $\mathcal{D}$, a set of relations $\mathcal{R}$, and a set of rules $\mathcal{Q}$. The variables, types, code locations, function names, etc. in the input program are mapped to integer values in their respective domains. Statements in the program are broken down into basic program operations. Each type of basic operation is represented by a relation; operations in a program are represented as tuples in corresponding input relations. A program analysis can declare additional domains and relations. Datalog rules define how the new domains and relations are computed.

A domain $D \in \mathcal{D}$ has size $size(D) \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. We require that all domains have finite size. The elements of a domain D is the set of natural numbers $0 \ldots size(D) - 1$.

A relation $R \in \mathcal{R}$ is a set of n-ary tuples of *attributes*. The kth attribute of relation R is signified by $a_k(R)$, and the number of attributes of a relation R is signified by $arity(R)$. Relations must have one or more attributes, i.e. $\forall R \in \mathcal{R}, arity(R) \geq 1$. Each attribute $a \in \mathcal{A}$ has a domain $domain(a) \in \mathcal{D}$, which defines the set of possible values for that attribute. An expression $R(x_1, \ldots, x_n)$ is true iff the tuple $(x_1, \ldots, x_n)$ is in relation R. Likewise, $\neg R(x_1, \ldots, x_n)$ is true iff $(x_1, \ldots, x_n)$ is *not* in R.

Rules are of the form:

$$E_0 :- E_1, \ldots, E_k.$$

where the expression E_0 (the rule *head*) is of the form $R(x_1, \ldots, x_n)$, where $R \in \mathcal{R}$ and $n = arity(R)$. The expression list $E_1, \ldots, E_k$ (the rule *subgoals*) is a list of zero or more expressions, each with one of the following forms:

- $R(x_1, \ldots, x_n)$, where $R \in \mathcal{R}$ and $n = arity(R)$
- $\neg R(x_1, \ldots, x_n)$, where $R \in \mathcal{R}$ and $n = arity(R)$
- $x_1 = x_2$
- $x_1 \neq x_2$
- $x_1 < x_2$

The comparison expressions $x_1 = x_2$, $x_1 \neq x_2$, and $x_1 < x_2$ have their normal meanings over the natural numbers.

The domain of a variable x is determined by its usage in a rule. If x appears as the kth argument of an expression of the form $R(x_1, \ldots, x_n)$ then the domain of x, denoted by $domain(x)$, is $domain(a_k(R))$. All uses of a variable within a rule must agree upon the domain. Furthermore, in a comparison expression such as $x_1 = x_2$, $x_1 \neq x_2$ or $x_1 < x_2$, the domains of variables x_1 and x_2 must match.

A *safe* Datalog program guarantees that the set of inferred facts (relation tuples) will be finite. In bddbddb, because all domains are finite, programs are necessarily safe. If a variable in the head of a rule does not appear in any subgoals, that variable may take on any value in the corresponding attribute's domain; i.e. it will be bound to the universal set for that domain.

bddbddb allows negation in *stratifiable* programs [11]. Rules are grouped into strata, which are solved in sequence. Each strata has a *minimal solution*, where relations have the minimum number of tuples necessary to satisfy those rules. In a stratified program, every negated predicate evaluates the negation of a relation which was fully computed in a previous strata.

Datalog with *well-founded* negation is a superset of Datalog with stratifiable negation, and can be used to express fixpoint queries [15]. We have not yet found it necessary to extend bddbddb to support well-founded semantics, though it would not be difficult.

2.2 Example

Algorithm 1 Context-insensitive points-to analysis with a precomputed call graph, where parameter passing is modeled with assignment statements.

DOMAINS

V	262144	`variable.map`
H	65536	`heap.map`
F	16384	`field.map`

RELATIONS

input	vP_0	$(\mathit{variable} : \mathrm{V}, \mathit{heap} : \mathrm{H})$
input	store	$(\mathit{base} : \mathrm{V}, \mathit{field} : \mathrm{F}, \mathit{source} : \mathrm{V})$
input	load	$(\mathit{base} : \mathrm{V}, \mathit{field} : \mathrm{F}, \mathit{dest} : \mathrm{V})$
input	assign	$(\mathit{dest} : \mathrm{V}, \mathit{source} : \mathrm{V})$
output	vP	$(\mathit{variable} : \mathrm{V}, \mathit{heap} : \mathrm{H})$
output	hP	$(\mathit{base} : \mathrm{H}, \mathit{field} : \mathrm{F}, \mathit{target} : \mathrm{H})$

RULES

$$vP(v,h) \quad :- \quad vP_0(v,h). \tag{1}$$
$$vP(v_1,h) \quad :- \quad \mathit{assign}(v_1,v_2), vP(v_2,h). \tag{2}$$
$$hP(h_1,f,h_2) \quad :- \quad \mathit{store}(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \tag{3}$$
$$vP(v_2,h_2) \quad :- \quad \mathit{load}(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2). \tag{4}$$

□

Algorithm 1 is the Datalog program for a simple Java points-to analysis. It begins with a declaration of domains, their sizes, and optional mapping files containing meaningful names for the numerical values in each domain. V is the domain of local variables and method parameters, distinguished by identifier name and lexical scope. H is the domain of heap objects, named by their allocation site. F is the domain of field identifiers, distinguished by name and the type of object in which they are contained.

Relations are declared next, along with the names and domains of their attributes. Relation vP_0 is the set of initial points-to relations. vP_0 is declared as a set of tuples (v,h), where $v \in$ V and $h \in$ H. $vP_0(v,h)$ is true iff the program directly places a reference to heap object h in variable v in an operation such as `s = new String()`. Relation store represents store operations such as `x.f = y`, and load similarly represents load operations. $\mathit{assign}(x,y)$ is true iff the program contains the assignment `x=y`. Assuming that a program call graph is available *a priori*, intraprocedural assignments from method invocation arguments to formal method parameters and assignments from return statements to return value destinations can be modeled as simple assignments.

The analysis infers possible points-to relations between heap objects, and possible points-to relations from variables to heap objects. $vP(v,h)$ is true if variable v may point to heap object h at any point during program execution. Similarly, $hP(h_1,f,h_2)$ is true if heap object field $h_1.f$ may point to heap object h_2.

Rule 1 incorporates the initial points-to relations into vP. Rule 2 computes the transitive closure over inclusion edges. If variable v_2 can point to object h and v_1 includes

v_2, then v_1 can also point to h. Rule 3 models the effect of store instructions on the heap. Given a statement $v_1.f = v_2$, if v_1 can point to h_1 and v_2 can point to h_2, then $h_1.f$ can point to h_2. Rule 4 resolves load instructions. Given a statement $v_2 = v_1.f$, if v_1 can point to h_1 and $h_1.f$ can point to h_2, then v_2 can point to h_2.

(a)	(b)
`String a = "fido";`	$vP_0(v_a, h_1)$
`String b;`	
`Dog d = new Dog();`	$vP_0(v_d, h_3)$
`b = a;`	$assign(v_b, v_a)$
`d.name = b;`	$store(v_d, name, v_b)$

Fig. 1. (a) Example program for Java pointer analysis. (b) Corresponding input relations.

To illustrate this analysis in action, we will use the simple Java program listed in Figure 1(a). Domain V contains values v_a, v_b and v_d representing variables `a`, `b`, and `d`. Domain H contains values h_1 and h_3, representing the objects allocated on lines 1 and 3. Domain F consists of the value $name$, which represents the `name` field of a `Dog` object.

The initial relations for this input program are given in Figure 1(b). Initial points-to relations in vP_0 are (v_a, h_1) and (v_d, h_3). The program has one assignment operation, represented as (v_b, v_a) in relation $assign$, and one store operation, represented as $(v_d, name, v_b)$ in relation $store$.

We begin by using Rule 1 to find that $vP(v_a, h_1)$ and $vP(v_d, h_3)$ are true. The results of the assignment on line 4 are found by using Rule 2, which tells us that $vP(v_b, h_1)$ is true since $assign(v_b, v_a)$ and $vP(v_a, h_1)$ are true. Finally, Rule 3 finds that $hP(h_3, name, h_1)$ is true, since $store(v_d, name, v_b)$, $vP(v_d, h_3)$, and $vP(v_b, h_1)$ are true.

3 From Datalog to BDD Operations

In this section, we explain our rationale for using BDD operations to solve Datalog programs. We first show how a Datalog program can be translated into relational algebra operations. We then show how we represent relations as boolean functions and relational algebra as operations on boolean functions. Finally, we show how boolean functions can be represented efficiently as binary decision diagrams (BDDs).

3.1 Relational Algebra

A Datalog query with finite domains and stratified negation can be solved by applying sequences of relational algebra operations corresponding to the Datalog rules iteratively, until a fixpoint solution is reached. We shall illustrate this translation simply by way of an example, since it is relatively well understood.

We use the following set of relational operations: join, union, project, rename, difference, and select. $R_1 \bowtie R_2$ denotes the *natural join* of relations R_1 and R_2, which returns a new relation where tuples in R_1 have been merged with tuples in R_2 in which corresponding attributes have equal values. $R_1 \cup R_2$ denotes the *union* of relations R_1 and R_2, which returns a new relation that contains the union of the sets of tuples in R_1 and R_2. $\pi_{a_1,\ldots,a_k}(R)$ denotes the *project* operation, which forms a new relation by removing attributes $a_1, \ldots, a_k$ from tuples in R. $\rho_{a \to a'}(R)$ denotes the *rename* operation, which returns a new relation with the attribute a of R renamed to a'. $R_1 - R_2$ denotes the *difference* of relations R_1 and R_2, which contains the tuples that are in R_1 but not in R_2. The *select* operation, denoted as $\sigma_{a=c}(R)$, restricts attribute a to match a constant value c. It is equivalent to performing a natural join with a unary relation consisting of a single tuple with attribute a holding value c.

To illustrate, an application of the rule

$$vP(v_1, h) : -assign(v_1, v_2), vP(v_2, h).$$

corresponds to this sequence of relational algebra operations:

$$\begin{array}{ll}
t_1 & = \rho_{variable \to source}(vP); \\
t_2 & = assign \bowtie t_1; \\
t_3 & = \pi_{source}(t_2); \\
t_4 & = \rho_{dest \to variable}(t_3); \\
vP & = vP \cup t_4;
\end{array}$$

Note that rename operations are inserted before join, union, or difference operations to ensure that corresponding attributes have the same name, while non-corresponding attributes have different names.

3.2 Boolean Functions

We encode relations as boolean functions over tuples of binary values. Elements in a domain are assigned consecutive numeric values, starting from 0. A value in a domain with m elements can be represented in $\lceil log_2(m) \rceil$ bits. Suppose each of the attributes of an n-ary relation R is associated with numeric domains $D_1, D_2, \ldots, D_n$, respectively. We can represent R as a boolean function $f : D_1 \times \ldots \times D_n \to \{0, 1\}$ such that $(d_1, \ldots, d_n) \in R$ iff $f(d_1, \ldots, d_n) = 1$, and $(d_1, \ldots, d_n) \notin R$ iff $f(d_1, \ldots, d_n) = 0$.

Let relation R be a set of tuples $\{(1, 1), (2, 0), (2, 1), (3, 0), (3, 1)\}$ over $D_1 \times D_2$, where $D_1 = \{0, 1, 2, 3\}$ and $D_2 = \{0, 1\}$. The binary encoding for R is function f, displayed in Figure 2(a), where the first attribute of R is represented by bits b_1 and b_2 and the second attribute by b_3.

For each relational algebra operation, there is a logical operation that produces the same effect when applied to the corresponding binary function representation. Suppose R_1 is represented by function $f_1 : D_1 \times D_2 \to \{0, 1\}$ and R_2 by function $f_2 : D_2 \times D_3 \to \{0, 1\}$. The relation $R_1 \bowtie R_2$ is represented by function $f_3 : D_1 \times D_2 \times D_3 \to \{0, 1\}$, where $f_3(d_1, d_2, d_3) = f_1(d_1, d_2) \wedge f_2(d_2, d_3)$. Similarly, the union operation maps to the binary $\vee$ operator, and $l - r \equiv l \wedge \neg r$. The project operation can be represented using existential quantification. For example, $\pi_{a_2}(R_1)$ is represented by $f : D_1 \to \{0, 1\}$ where $f(d_1) = \exists d_2.f_1(d_1, d_2)$.

3.3 Binary Decision Diagrams

Large boolean functions can be represented efficiently using BDDs, which were originally invented for hardware verification to efficiently store a large number of states that share many commonalities [8].

A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes which represent the constants one and zero. This graph represents a boolean function over a set of input decision variables. Each non-terminal node t in the DAG is labeled with an input decision variable and has exactly two outgoing edges: a high edge and a low edge. To evaluate the function for a given set of input values, one simply traces a path from the root node to one of the terminal nodes, following the high edge of a node if the corresponding input variable is true, and the low edge if it is false. The terminal node gives the value of the function for that input. Figure 2(b) shows a BDD representation for function f from Figure 2(a). Each non-terminal node is labeled with the corresponding decision variable, and a solid line indicates a high edge while a dashed line indicates a low edge.

D_1		D_2	R
b_1	b_2	b_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a)

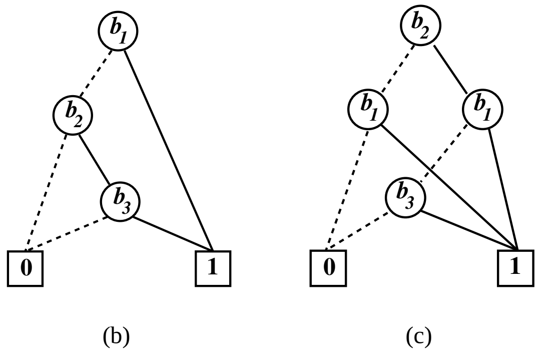

Fig. 2. (a) Binary encoding of a relation. (b) and (c) are BDD encodings of the relation given by (a) with decision variable orders b_1, b_2, b_3 and b_2, b_1, b_3, respectively.

We specifically use a variant of BDDs called *reduced ordered binary decision diagrams*, or ROBDDs [8]. In an *ordered* BDD, the sequence of variables evaluated along any path in the DAG is guaranteed to respect a given total *decision variable order*. The choice of the decision variable order can significantly affect the number of nodes required in a BDD. The BDD in Figure 2(b) uses variable order b_1, b_2, b_3, while the BDD in Figure 2(c) represents the same function, only with variable order b_2, b_1, b_3. Though the change in order only adds one extra node in this example, in the worst case an exponential number of nodes can be added. In addition, ROBDDs are *maximally reduced* meaning common BDD subgraphs are collapsed into a single graph, and the nodes are shared. Therefore, the size of the ROBDD depends on whether there are common boolean subexpressions in the encoded function, rather than on the number of entries in the set.

3.4 BDD Operations

The boolean function operations discussed in Section 3.2 are a standard feature of BDD libraries [20]. The $\wedge$ (and), $\vee$ (or), and $-$ (difference) boolean function operations can be applied to two BDDs, producing a BDD of the resulting function. The BDD existential quantification operation `exist` is used to produce a new BDD where nodes corresponding to projected attributes are removed. This operation combines the low and high successors of each removed node by applying an $\vee$ operation.

Rename operations are implemented using the BDD `replace` operation, which computes a new BDD where decision variables corresponding to the old attributes have been replaced with decision variables corresponding to the new attribute names. Replace operations can be eliminated if the renamed attributes are encoded using the same decision variables as the original attributes. A `replace` operation which does not change the relative order of decision variables is only linear with respect to the number of nodes in the BDD. If the order is changed, the cost of a `replace` can be exponential with respect to the number of decision variables. Care must be taken when encoding relation attributes to minimize the number of expensive rename operations.

Natural join operations are frequently followed by project operations to eliminate unnecessary attributes. The BDD relational product operation, or `relprod`, efficiently combines this sequence in a single operation. Similarly, the select and project operations can be combined into a single BDD operation, known as `restrict`.

BDD operations operate on entire relations at a time, rather than one tuple at a time. The cost of BDD operations depends on the size and shape of the BDD graphs, not the number of tuples in a relation. Thus, large relations can be computed quickly as long as their encoded BDD representations are compact. Also, due to caching in BDD packages, identical subproblems only have to be computed once. These points are key to the efficiency of BDD operations, and are the reason why we use this data structure to represent our relations.

4 Translating and Optimizing Datalog Programs

The bddbddb system applies a large number of optimizations to transform Datalog programs into efficient BDD operations:

1. Apply Datalog source level transforms and optimizations. (Section 4.1)
2. Remove unnecessary rules, stratify the rules, and determine the rule iteration order. (Section 4.2)
3. Translate the stratified query into an intermediate representation (IR) consisting of relational algebra operations. (Section 4.3)
4. Through analysis, optimize the IR and add BDD operations to replace equivalent sequences of relational algebra operations. (Section 4.4)
5. Choose BDD decision variables for encoding relation attributes. (Section 4.5)
6. Perform more dataflow optimizations after physical domains have been assigned. (Section 4.6)
7. Interpret the resulting program. (Section 4.7)

To illustrate this process, we use Algorithm 1 from Section 2 as a running example.

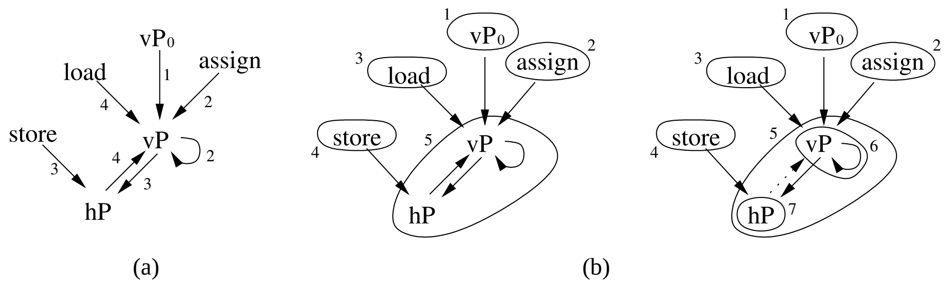

Fig. 3. (a) Predicate dependency graph for Algorithm 1. (b) Breaking the PDG into SCCs and finding cycles.

4.1 Datalog Source Transformations

Before compilation, we normalize the forms of the input rules as follows:

– Any variable that appears only once in a rule is changed into an underscore (_) to indicate an unused attribute.
– If a variable appears multiple times in a single subgoal, we give each additional use a distinct name, and then add extra equality subgoals to make the new variables equal to the original variable. For example, a subgoal $R(x, x, x)$ is transformed into the three subgoals $R(x, x', x''), x = x', x = x''$.
– Each comparison subgoal with an attribute of domain D is substituted with a subgoal for a corresponding precomputed relation defined over $D \times D$ which represents that comparison function.
– Subgoals in rules that define temporary relations are inlined into the rules that use those relations. Temporary relations are non-input, non-output relations which are in the head of only one rule, and appear as a subgoal in only one other rule.

4.2 Datalog Rule Optimization

Rule Removal The solver removes rules and relations that do not indirectly contribute to the output relations. A *predicate dependency graph* (PDG) is built to record dependencies between rules and relations. Each node represents a relation, and there is an edge $g \rightarrow h$ marked with rule r if rule r has subgoal relation g and head relation h. (If the subgoal is negated, the edge is marked as a negative edge.) The PDG for our example is shown in Figure 3(a). Necessary rules and relations are found by performing a backward pass over the PDG, starting from the output relations.

Stratification We then use the PDG to stratify the program. Stratification guarantees that the relation for every negated subgoal can be fully computed before applying rules containing the negation. Each stratum is a distinct subset of program rules that fully computes relations belonging to that stratum. Rules in a particular stratum may use the positive forms of relations computed in that stratum, as well as positive or negated forms of relations calculated in earlier strata and input relations from the relational database. There are no cyclic dependencies between strata. If the program cannot be stratified, we warn the user. In our experience designing Datalog programs for program analysis, we have yet to find a need for non-stratifiable queries.

As our example does not contain any negations, all of the rules and relations are placed within a single stratum.

Finding cycles Cycles in the PDG indicate that some rules and relations are recursively defined, requiring iterative application of rules within the cycles to reach a fixed-point solution. The PDG for each stratum is split into strongly connected components (SCCs). We can compute the result for a stratum by evaluating strongly connected components and non-cyclic relations in the topological order of the PDG.

A single strongly connected component can encompass multiple loops that share the same header node. We would like to distinguish between the different loops in a single SCC so we can iterate around them independently. However, the PDG is typically not reducible, and the classical algorithm for finding loops—Tarjan's interval finding algorithm—only works on reducible graphs [34]. Extensions have been made to deal with irreducible graphs, but they typically have the property that a node can only be the header for one loop [28]. We solve this by identifying one loop in the SCC, eliminating its back edge, and then recursively re-applying the SCC algorithm on the interior nodes to find more inner loops.

The steps of the algorithm on our example are shown in Figure 3(b). We first break the PDG into five SCCs, labeled 1-5, as shown on the left. Then, we remove the edge for rule 4 from *hP* to *vP*, breaking the larger cycle so that it can topologically sort those nodes and find the smaller self-cycle on *vP* for rule 2, as shown on the right.

Determining rule application order The order in which the rules are applied can make a significant difference in the execution time. When there are multiple cycles in a single SCC, the number of rule applications that are necessary to reach a fixpoint solution can differ based on the relative order in which the two cycles are iterated. Which application order will yield the fewest number of rule applications depends not only on the rules but also on the nature of the relations.

Aspects of the BDD library can also make certain iteration orders more efficient than others, even if they have more rule applications. For example, the BDD library uses an operation cache to memoize the results of its recursive descents on BDD nodes, so it can avoid redundant computations when performing an operation. This cache can also provide benefits across different operations if the BDDs that are being operated upon share nodes. To take advantage of operation cache locality across operations, one should perform related operations in sequence. Another aspect influencing iteration order choice is the set-oriented nature of BDD operations. When performing an operation on tuples generated in a loop, it is ofter faster to apply the operation after completing all loop iterations, rather than applying it once per loop iteration.

In the absence of profile information from prior runs or from the user, bddbddb uses static analysis of the rules to decide upon a rule application order. Cycles that involve fewer rules are iterated before cycles that involve more rules, and rules that have fewer subgoals are iterated before rules that have more subgoals. The reasoning behind this is that smaller, shorter chains of rules and smaller rules are faster to iterate due to operation cache locality. This static metric works very well in the examples we have tried because small cycles are usually transitive closure computations, which are fast and expose more opportunities for set-based computation on the larger cycles.

4.3 Intermediate Representation

Once we have determined the iteration order, we translate the rules into an intermediate representation based on relational algebra operations as follows:

1. For each subgoal with an underscore, project away its unused attributes.
2. For each subgoal with a constant, use the select and project operators to restrict the relation to match the constant.
3. Join each subgoal relation with each of the other subgoal relations, projecting away attributes as they become unnecessary.
4. Rename the attributes in the result to match the head relation.
5. If the head relation contains a constant, use the select operator on the result to set the value of the constant.
6. Unify the result with the head relation.

4.4 IR Optimizations

In repeated applications of a given rule within a loop, it can be more efficient to make use of the differential between the current value of a subgoal relation and the previous value from the last time the rule was applied. This is known as *incrementalization* or the *semi-naïve* evaluation strategy. By computing the difference in subgoal relations as compared to the previous iteration, we can avoid extra work; if these inputs are the same as the last iteration, we can avoid applying the rule altogether. The tradeoff of incrementalization is that the old value of every subgoal in every incrementalized rule must be stored. We allow the user to control whether incrementalization is performed on a per rule basis. Performing incrementalization on the sequence of relational algebra operations derived from Rule (2) (Section 3.1) generates the following IR:

$$
\begin{aligned}
vP'' &= vP - vP';\\
vP' &= vP;\\
assign'' &= assign - assign';\\
assign' &= assign;\\
t_1 &= \rho_{variable\rightarrow source}(vP'');\\
t_2 &= assign \bowtie t_1;\\
t_3 &= \rho_{variable\rightarrow source}(vP);\\
t_4 &= assign'' \bowtie t_3;\\
t_5 &= t_2 \cup t_4;\\
t_6 &= \pi_{source}(t_5);\\
t_7 &= \rho_{dest\rightarrow variable}(t_6);\\
vP &= vP \cup t_7;
\end{aligned}
$$

Next, we apply a number of traditional compiler data flow optimizations on the IR:

– **Constant propagation**. We propagate empty set, universal set, and constants to reduce unions, joins, and difference operations.
– **Definition-use chains**. We calculate the chains of definitions and uses and use this to optimize the program by eliminating dead code (operations whose results have no uses), coalescing natural join and project pairs into `relprod` operations, and coalescing select and project pairs into `restrict` operations.

After this stage of optimizations, relational algebra operations are replaced by BDD operations, using combined `relprod` operations and `restrict` operations where possible. Rule (2) becomes:

$$\begin{aligned}
vP'' &= \texttt{diff}(vP, vP');\\
vP' &= \texttt{copy}(vP);\\
t_1 &= \texttt{replace}(vP'', \mathit{variable} \rightarrow \mathit{source});\\
t_2 &= \texttt{relprod}(t_1, \mathit{assign}, \mathit{source});\\
t_3 &= \texttt{replace}(t_2, \mathit{dest} \rightarrow \mathit{variable});\\
vP &= \texttt{or}(vP, t_3);
\end{aligned}$$

In the optimized IR, the join-project pair involving $assign$ and vP'' has been collapsed into a single `relprod`. Also, the operations for computing and using the difference of $assign$ have been removed because $assign$ is loop invariant.

4.5 BDD Decision Variable Assignment

As noted in Section 3.4, the use of BDD operations to implement relational operations places constraints on the choice of BDD decision variables used to encode relation attributes. When performing an operation on two BDDs, the decision variables for corresponding attributes must match. Likewise, unmatched attributes must be assigned to different decision variables. A BDD `replace` operation is used whenever different sets of decision variables must be substituted into a BDD as the result of a relational rename.

It is most important to minimize the cost of `replace` operations. This depends on the choice of decision variables used for encoding each attribute. The cost can be zero, linear, or exponential depending on whether the new decision variables are the same, have the same relative order, or have a different relative order. Additionally, we prefer to perform costly `replace` operations on smaller BDDs (in terms of BDD nodes) rather than on larger BDDs.

bddbddb uses a priority-based constraint system to assign attributes to BDD decision variables. This system is expressed in terms of both equivalence and non-equivalence constraints on relation attributes and sequences of decision variables. We use a specialized union-find data structure augmented with non-equivalence constraints to efficiently compute the constraint system. In BDD terminology, a sequence of binary decision variables used to represent an attribute is often referred to as a *physical domain*, which should not be confused with a Datalog domain as defined in Section 2.1.

We avoid introducing `replace` operations by constraining any renamed attributes to use the same physical domain as the original attribute. When an introduced constraint would cause the constraint system to have no solution, we assign the new attribute to a different physical domain and add a `replace` operation at that point to allow the constraint to be satisfied. By carefully choosing the order of priority in which constraints are added to the system, we ensure that `replace` operations are introduced where they will be most efficient.

For each attribute a in relation R, we create a non-equivalence constraint between a and other attributes in R. Then, we add constraints for all program operations, in order of importance. Operations in inner loops have higher importance than operations in outer loops, under the presumption that these operations will be performed more often. Within a given loop depth, `relprod` operations are considered first, in order of execution, since they are typically the most expensive operations. After `relprod` operations, we consider other operations. For a unary operation such as `copy`, we create equivalence constraints between corresponding attributes of the source and destination relations. For a binary operation, the interacting attributes for the input relations are constrained to be equal. After considering all operations, we add constraints for the attributes of the input and output relations. The physical domains used by these relations are specified by the user, since they must be loaded from or stored into the relational database.

An application of the physical domain assignment algorithm to our running example reveals that $variable$ from vP'' and $source$ from $assign$ can be assigned to the same physical domain for the `relprod`. Therefore, the `replace` that occurs immediately before can be removed:

$$\begin{aligned}
vP'' &= \texttt{diff}(vP, vP');\\
vP' &= \texttt{copy}(vP);\\
t_1 &= \texttt{relprod}(vP'', assign, source);\\
t_2 &= \texttt{replace}(t_1, dest[V1] \rightarrow variable[V0]);\\
vP &= \texttt{or}(vP, t_2);
\end{aligned}$$

4.6 Additional Optimizations

After domain assignment, we have the opportunity to apply another set of standard compiler optimizations:

- **Global value numbering**. Global value numbering factors the evaluation of common subexpressions among rules into non-redundant computations. Moreover, it optimizes loops by hoisting invariants.
- **Copy propagation**. Copy propagation eliminates unnecessary temporary IR relations that can be generated by our optimizations.
- **Liveness analysis**. We use a liveness analysis to clean up dead code. We reduce the memory footprint during IR interpretation by freeing relation allocations as soon as the lifetime of a relation has ended.

4.7 Interpretation

Finally, bddbddb interprets the optimized IR and performs the IR operations in sequence by calling the appropriate methods in the BDD library.

4.8 Decision Variable Ordering

While BDDs have proven effective in compacting the commonalities in large sets of data, the extent to which these commonalities can be exploited depends on the ordering of the decision variables. In our case, the difference between a good or bad ordering can mean the termination or non-termination (due to memory exhaustion) of an analysis. Moreover, the relative orderings are not readily apparent given only a static analysis, and the space of all orders is extremely large (with both precedence and interleaving conditions, the number of orders is given by the series for ordered Bell numbers).

We have developed an algorithm for finding an effective decision variable ordering [9]. The algorithm, based on active learning, is embedded in the execution of Datalog programs in the bddbddb system. When bddbddb encounters a rule application that takes longer than a parameterized amount of time, it initiates a learning episode to find a better decision variable ordering by measuring the time taken for alternative variable orderings. Because rule applications can be expensive, bddbddb maximizes the effectiveness of each trial by actively seeking out those decision variable orderings whose effects are least known.

5 Experimental Results

We measure the effectiveness of our bddbddb system and compare it to hand-optimized BDD programs. Prior to developing the bddbddb system, we had manually implemented and optimized three points-to analyses: a context-insensitive pointer analysis for Java described by Berndl [5], a context-sensitive pointer analysis based on the cloning of paths in the call graph [38], and a field-sensitive, context-insensitive pointer analysis for C [1]. We then wrote Datalog versions of these analyses which we ran using the bddbddb system.

The hand-coded Java analyses are the result of months of effort and are well-tuned and optimized. The variable ordering and physical domain assignment have been carefully hand-tuned to achieve the best results. Many of the rules in the hand-coded algorithms were incrementalized. This proved to be a very tedious and error-prone process, and we did not incrementalize the whole system as it would have been too unwieldy. Bugs were still popping up weeks after the incrementalization was completed. bddbddb, on the other hand, happily decomposed and incrementalized even the largest and most complex inference rules.

Because of the unsafe nature of C, the C pointer analysis is much more complicated, consisting of many more rules. For the hand-coded C pointer analysis, physical domain assignments, domain variable orderings and the order of inferences were only optimized to avoid significant execution slowdowns. Specification of low-level BDD operations was an error-prone, time-consuming process. A good deal of time was spent modifying physical domain assignments and solving errors due to the incorrect specification of physical domains in BDD operations. Once the Datalog version of the analysis was specified, development of the hand-coded version was discontinued, as it was no longer worth the effort. In the experiment reported here, we compare the hand-coded version and equivalent Datalog implementation from that time.

We also evaluate the performance of bddbddb on two additional analyses: an analysis to find external lock objects to aid in finding data races and atomicity bugs, and an analysis to find SQL injection vulnerabilities in Java web applications [23]. Both of these analyses build on top of the context-sensitive Java pointer analysis, and both are fairly sophisticated analyses. We do not have hand-coded implementations of these analyses as they would be too tedious to implement by hand.

5.1 Comparing Lines of Code

The first metric for comparison is in the number of lines of code in each algorithm:

Analysis	Hand-coded	Datalog
context-insensitive Java	1975	30
context-sensitive Java	3451	33
context-insensitive C	1363	308
external lock analysis	n/a	42
SQL injection analysis	n/a	38

Fig. 4. LOC for hand-coded analyses versus lines of Datalog using bddbddb

Specifying the analysis as Datalog reduced the size of the analysis by 4.4 times in the case of the C analysis, to over 100 times in the case of the context-sensitive Java analysis. The disparity between the C and Java implementations is due to the fact that the C implementation combined many BDD operations on a single line, whereas the Java implementation put each BDD operation on a separate line of code.

Adding a new analysis with bddbddb takes only a few lines of code versus a rewrite of thousands of lines for a hand-coded implementation. The external lock analysis and the SQL injection analysis are examples of this. In another example, we easily modified the inference rules for the context-insensitive C points-to analysis to create a context-sensitive analysis by adding an additional context attribute to existing relations. While this was an extremely simple change to make to the bddbddb Datalog specification, such a modification would have required rewriting hundreds of lines of low-level BDD operations in the hand-coded analysis.

5.2 Comparing Analysis Times

For each analysis, we compared the solve time for an incrementalized hand-coded implementation against a bddbddb-based implementation with varying levels of optimization. Analyses were performed on an AMD Opteron 150 with 4GB RAM running RedHat Enterprise Linux 3 and Java JDK 5.0. The three bddbddb-based analyses and the hand-coded Java points-to analysis used the open-source JavaBDD library [37], which internally makes use of the BuDDy BDD library [20]. The hand-coded C points-to analysis makes direct use of the BuDDy library. The Java context-insensitive analysis used an initial node table size of 5M and an operation cache size of 750K. The Java context-sensitive analysis and C points-to analyses both used an initial node table size of 10M and an operation cache size of 1.5M.

Figures 5, 6 and 7 contain the run times of our Java context-insensitive analysis, Java context-sensitive analysis, and C pointer analysis, respectively. The first two columns give the benchmark name and description. The next column gives the solve time in seconds for the hand-coded solver. The remaining columns give the solve time when using bddbddb with various optimizations enabled. Each column adds a new optimization in addition to those used in columns to the left. Under **No Opts** we have all optimizations disabled. Under **Incr** we add incrementalization, as described in Section 4.3. Under **+DU** we add optimizations based on definition-use chains. Under **+Dom** we optimize physical domain assignments. Under **+All** we add the remaining optimizations described in Section 4. For the Java context-insensitive and C pointer analyses, the **+Order** column shows the result of bddbddb with all optimizations enabled using a variable order discovered by the learning algorithm referred to in Section 4.8. For our C programs, we used the order learned from `enscript`. For the Java programs we used the order learned from `joeq`. In the Java context-sensitive case, the learning algorithm

Name	Description	Hand-coded	bddbddb No Opts	Incr	+DU	+Dom	+All	+Order
joeq	virtual machine and compiler	7.3	10.0	9.4	7.9	4.8	4.5	3.6
jgraph	graph-theory library	15.0	25.6	24.1	20.0	11.0	10.4	7.6
jbidwatch	auction site tool	26.3	47.4	45.8	35.4	18.6	16.8	13.0
jedit	sourcecode editor	67.0	123.5	119.9	100.0	56.4	45.7	35.7
umldot	UML class diagrams from Java	16.6	29.0	27.4	20.2	11.6	10.9	8.4
megamek	networked battletech game	35.8	71.3	67.1	57.0	26.8	23.0	17.4

Fig. 5. Comparison of context-insensitive Java pointer analysis runtimes. Times are in seconds.

Name	Description	Hand-coded	bddbddb No Opts	Incr	+DU	+Dom	+All
joeq	virtual machine and compiler	85.3	323.3	317.8	274.7	124.7	69.7
jgraph	graph-theory library	118.0	428.1	431.1	362.2	116.3	94.9
jbidwatch	auction site tool	421.1	1590.2	1533.3	1324.3	470.6	361.3
jedit	sourcecode editor	147.0	377.2	363.4	293.7	136.4	109.3
umldot	UML class diagrams from Java	402.5	1548.3	1619.3	1362.3	456.5	332.8
megamek	networked battletech game	1219.2	∞	∞	4306.5	1762.9	858.3

Fig. 6. Comparison of context-sensitive Java pointer analysis runtimes. Times are in seconds.

Name	Description	Hand-coded	bddbddb No Opts	Incr	+DU	+Dom	+All	+Order
crafty	chess program	8.7	547.3	525.9	571.7	9.4	8.1	8.2
enscript	text to PS conversion	41.0	1175.4	1211.7	1128.4	122.3	112.6	31.5
hypermail	mbox to HTML conversion	149.4	6263.8	6113.0	5967.1	262.0	231.3	44.2
monkey	webserver	16.9	468.9	397.7	398.7	33.1	31.3	9.6

Fig. 7. Comparison of C pointer analysis runtimes. Times are in seconds.

Name	Description	bddbddb No Opts	Incr	+DU	+Dom	+All
joeq	virtual machine and compiler	75.0	60.4	59.3	17.4	15.1
jgraph	graph-theory library	64.9	51.0	51.1	13.0	12.5
jbidwatch	auction site tool	231.0	183.6	203.5	52.3	51.7
jedit	sourcecode editor	20.1	16.3	16.2	5.3	5.1
umldot	UML class diagrams from Java	199.3	162.2	161.3	45.0	39.2
megamek	networked battletech game	13.3	11.5	10.5	5.1	4.3

Fig. 8. External lock analysis runtimes. Times are in seconds.

Name	Description	bddbddb No Opts	Incr	+DU	+Dom	+All	+Order
personalblog	J2EE-based blogging application	∞	73.0	57.8	25.1	23.1	16.7
road2hibernate	hibernate testing application	∞	86.4	74.8	49.2	39.7	33.4
snipsnap	J2EE-based blogging application	∞	227.8	211.9	98.9	84.5	55.8
roller	J2EE-based blogging application	∞	521.0	479.0	253.7	208.4	185.4

Fig. 9. SQL injection query results. Times are in seconds. ∞ indicates that the analysis did not finish.

was not able to find a better order, so we omitted this column. Entries marked with a ∞ signified that the test case did not complete due to running out of memory.

The time spent by bddbddb to translate Datalog to optimized BDD operations is negligible compared to the solve times, so the translation times have been omitted. In all cases, bddbddb spent no more than a few seconds to compile the Datalog into BDD operations.

The unoptimized context-insensitive Java analysis was 1.4 to 2 times slower than the hand-coded version. Incrementalization showed a very small improvement, but by adding def-use optimizations, we began to see a useful time reduction to 80% of the original. Optimizing BDD domain assignments reduces the runtime to about 42% of the original, and enabling all optimizations further reduces the runtime to about 38% of the original. Improved variable order brought the runtime between 24% and 36% of the unoptimized runtime. While incrementalization and def-use optimizations were sufficient to bring the bddbddb analysis close to the hand-coded analysis runtimes, the remaining optimizations and learned variable order combined to beat the hand-coded solver runtime by a factor of 2.

Results for the context-sensitive Java analysis were similar to the context-insensitive results. Unfortunately, our variable order learning algorithm was unable to learn a better variable order for this analysis, leaving the fully optimized bddbddb analysis about 20% faster than the hand-coded version.

In the case of the C analysis, the unoptimized bddbddb analysis was 23 to 60 times slower than the hand-coded version. This is likely due to the relative complexity of the Datalog in the C analysis case; optimizations were able to make significant improvements to the execution times. Analysis times with all optimizations enabled were roughly comparable to our hand-coded solver. As with the Java analyses, the largest gain was due to optimized physical domain assignment. When applying the learned variable order, bddbddb analysis runtimes were reduced even further, to fall between 30% and 95% of the hand-coded implementation.

5.3 External Lock and SQL Injection Analyses

We also used bddbddb to build external lock and SQL injection detection analyses on top of the Java points-to analysis results. The runtimes for the external lock analysis using different levels of optimization are displayed in Figure 8. Incrementalization reduces the analysis time to about 80% of the original time. Optimizing physical domain assignments further reduces the analysis time to about 23% of the original. Figure 9 displays the runtimes of the SQL injection analysis on four web-based applications. Without any incrementalization, the analysis fails to complete due to memory exhaustion. However, with further optimization we see performance gains similar to those of the external lock analysis.

6 Related Work

Related work falls into three general categories: optimizing Datalog executions, logic programming systems that use BDDs, and program analysis with BDDs. We go through each category in turn.

6.1 Optimizing Datalog

Liu and Stoller described a method for transforming Datalog rules into an efficient implementation based on indexed and linked data structures [21]. They proved their technique has "optimal" run time with respect to the fact that the combinations of facts that lead to all hypotheses of a rule being simultaneously true are considered exactly once. They did not present experimental results. Their formulation also greatly simplified the complexity analysis of Datalog programs. However, their technique does not apply when using BDDs, as the cost of BDD operations does not depend upon combinations of facts, but rather upon the number of nodes in the BDD representation and the nature of the relations.

There has been lots of research on optimizing Datalog evaluation strategies; for example, semi-naïve evaluation [10], bottom-up evaluation [10, 24, 35], top-down with tabling [12, 33], etc. Ramakrishnan et al. investigated the role of rule ordering in computing fixpoints [26]. We use an evaluation strategy geared towards peculiarities of the BDD data structure — for example, to maximize cache locality, we iterate around inner loops first.

There has been work on transforming Datalog programs to reduce the amount of work necessary to compute a solution. Magic sets is a general algorithm for rewriting logical rules to cut down on the number of irrelevant facts generated [4]. This idea was extended to add better support for certain kinds of recursion [25]. Sagiv presented an algorithm for optimizing a Datalog program under uniform equivalence [30]. Zhou and Sato present several optimization techniques for fast computation of fixpoints by avoiding redundant evaluation of subgoals [39].

Halevy et al. describe the *query-tree*, a data structure that is useful in the optimization of Datalog programs [16]. The query-tree encodes all symbolic derivation trees that satisfy some property.

6.2 Logic Programming with BDDs

Iwaihara et al. described a technique for using BDDs for logic programming [17]. They presented two different ways of encoding relations: logarithmic encoding, which is the encoding we use in this paper, and linear encoding, which encodes elements or parts of elements as their own BDD variable. They evaluate the technique using a transitive closure computation. The Toupie system translates logic programming queries into an implementation based on decision diagrams [13].

Crocopat is a tool for relational computation that is used for structural analysis of software systems [7]. Like bddbddb, they use BDDs to represent relations.

6.3 Program Analysis with BDDs

Both Zhu and Berndl et al. used BDDs to implement context-insensitive inclusion-based points-to analysis [5, 40]. Zhu extended his technique to support a summary-based context sensitivity [41], whereas Whaley and Lam developed a cloning-based context-sensitive pointer analysis algorithm that relies heavily on the data sharing inherent in BDDs [38]. Avots et al. extended Whaley and Lam's algorithm to support C programs with pointer arithmetic [1].

Jedd is a Java language extension that provides a relational algebra abstraction over BDDs [19]. Their treatment of domain assignment as a constraint problem is similar to ours; they use a SAT solver to find a legal domain assignment. They do not attempt to order the constraints based on importance.

Besson and Jensen describe a framework that uses Datalog to specify a variety of class analyses for object oriented programs [6]. Sittampalam, de Moor, and Larsen formulate program analyses using conditions on control flow paths [32]. These conditions contain free metavariables corresponding to program elements (such as variables and constants). They use BDDs to efficiently represent and search the large space of possible instantiations.

Bebop is a symbolic model checker used for checking program behavior [2]. It uses BDDs to represent sets of states. It has been used to validate critical safety properties of device drivers [3].

7 Conclusion

This paper described bddbddb, a deductive database engine that uses Datalog for specifying and querying program analyses. Datalog is a natural means of specifying many program analyses; many complicated analyses can be specified in only a few lines of Datalog. Adding BDDs to this combination works well because BDDs can take advantage of the redundancies that occur in program analyses — especially context-sensitive analyses — and because BDD whole-set operations correspond closely to Datalog's evaluation style.

Our experience with the system is encouraging. Program analyses are so much easier to implement using bddbddb that we can no longer go back to the old technique of hand coding analyses. This is especially true because our experiments showed that bddbddb can often execute program analyses faster than a well-tuned handcoded implementation. Although there is still much work to be done in improving the algorithms and implementation of bddbddb, we have found the tool to be useful in our research.

The use of our system brings many benefits. It makes prototyping new analyses remarkably easy. Combining the results of multiple analyses becomes trivial. Concise specifications are easier to verify than larger traditional programs. The analysis runs faster because the inference engine automates the tedious process of optimizing and incrementalizing the analysis. New optimizations can be tested and implemented once in the inference engine, rather than repeatedly in each analysis. bddbddb bridges the gap between the specification of a program analysis and its implementation.

However, the greatest benefit of our system is that it makes powerful program analysis more widely accessible. The ease of a declarative language like SQL is considered to be one of the reasons for the success in databases [27]. We believe that the use of Datalog may play a important role in the future of interactive programming tools.

The bddbddb system is publicly available on Sourceforge licensed under the open-source LGPL license.

Acknowledgments

The authors would like to thank V. Benjamin Livshits and Christopher Unkel for their contribution of test cases and helpful feedback on their experiences with using bddbddb. This work was supported in part by the NSF under Grant No. 0326227 and an Intel Graduate Fellowship.

References

1. D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. ACM Press, 2005.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, 2000.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122. Springer-Verlag New York, Inc., 2001.
4. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15. ACM Press, 1986.
5. M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.
6. F. Besson and T. Jensen. Modular class analysis with datalog. In R. Cousot, editor, *Proceedings of the 10th Static Analysis Symposium (SAS 2003)*, pages 19–36. Springer LNCS vol. 2694, 2003.
7. D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, Nov. 2003.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
9. M. Carbin, J. Whaley, and M. S. Lam. Finding effective variable orderings for BDD-based program analysis. To be submitted for publication, 2005.
10. S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., 1990.
11. A. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.
12. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
13. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. L. Charlier. Efficient bottom-up abstract interpretation of prolog by means of constraint solving over symbolic finite domains. In *PLILP '93: Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 75–91. Springer-Verlag, 1993.
14. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systemsa case study. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
15. A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.
16. A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in datalog extensions. *J. ACM*, 48(5):971–1012, 2001.
17. M. Iwaihara and Y. Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 467–474. IEEE Computer Society, 1995.
18. M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, June 2005.

19. O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 158–169. ACM Press, 2004.
20. J. Lind-Nielsen. BuDDy, a binary decision diagram package. http://buddy.sourceforge.net.
21. Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183. ACM Press, 2003.
22. V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium*. USENIX, Aug. 2005.
23. M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2005.
24. J. F. Naughton and R. Ramakrishnan. Bottom-up evaluation of logic programs. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 640–700, 1991.
25. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 235–242. ACM Press, 1989.
26. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 359–371. Morgan Kaufmann Publishers Inc., 1990.
27. R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *J. Logic Programming*, 23(2):125–149, 1993.
28. G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, Mar. 1999.
29. T. W. Reps. *Demand Interprocedural Program Analysis Using Logic Databases*, pages 163–196. Kluwer, 1994.
30. Y. Sagiv. Optimizing datalog programs. In *PODS '87: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 349–362. ACM Press, 1987.
31. K. Sagonas, T. Swift, and D. S. Warren. Xsb as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 442–453. ACM Press, 1994.
32. G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38. ACM Press, 2004.
33. H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings on Third International Conference on Logic Programming*, pages 84–98. Springer-Verlag New York, Inc., 1986.
34. R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, Dec. 1974.
35. J. D. Ullman. Bottom-up beats top-down for datalog. In *PODS '89: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 140–149. ACM Press, 1989.
36. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, MD., volume II edition, 1989.
37. J. Whaley. JavaBDD library. http://javabdd.sourceforge.net.
38. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144. ACM Press, 2004.
39. N.-F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 275–283. ACM Press, 2003.
40. J. Zhu. Symbolic pointer analysis. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 150–157. ACM Press, 2002.
41. J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157. ACM Press, 2004.