Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams

John Whaley Monica S. Lam
Computer Science Department
Stanford University
Stanford, CA 94305
{jwhaley, lam}@stanford.edu

ABSTRACT

This paper presents the first scalable context-sensitive, inclusion-based pointer alias analysis for Java programs. Our approach to context sensitivity is to create a clone of a method for every context of interest, and run a *context-insensitive* algorithm over the expanded call graph to get *context-sensitive* results. For precision, we generate a clone for every acyclic path through a program's call graph, treating methods in a strongly connected component as a single node. Normally, this formulation is hopelessly intractable as a call graph often has 10^{14} acyclic paths or more. We show that these exponential relations can be computed efficiently using binary decision diagrams (BDDs). Key to the scalability of the technique is a context numbering scheme that exposes the commonalities across contexts. We applied our algorithm to the most popular applications available on Sourceforge, and found that the largest programs, with hundreds of thousands of Java bytecodes, can be analyzed in under 20 minutes.

This paper shows that pointer analysis, and many other queries and algorithms, can be described succinctly and declaratively using Datalog, a logic programming language. We have developed a system called bddbddb that automatically translates Datalog programs into highly efficient BDD implementations. We used this approach to develop a variety of context-sensitive algorithms including side effect analysis, type analysis, and escape analysis.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; E.2 [**Data**]: Data Storage Representations

General Terms

Algorithms, Performance, Design, Experimentation, Languages

Keywords

context-sensitive, inclusion-based, pointer analysis, Java, scalable, cloning, binary decision diagrams, program analysis, Datalog, logic programming

PLDI'04, June 9–11, 2004, Washington, DC, USA.

1. INTRODUCTION

Many applications of program analysis, such as program optimization, parallelization, error detection and program understanding, need pointer alias information. Scalable pointer analyses developed to date are imprecise because they are either *context-insensitive*[3, 17, 19, 33] or *unification-based*[15, 16]. A context-insensitive analysis does not distinguish between different calling contexts of a method and allows information from one caller to propagate erroneously to another caller of the same method. In unification-based approaches, pointers are assumed to be either unaliased or are pointing to the same set of locations[28]. In contrast, *inclusion-based* approaches are more efficient but also more expensive, as they allow two aliased pointers to point to overlapping but different sets of locations.

We have developed a context-sensitive and inclusion-based pointer alias analysis that scales to hundreds of thousands of Java bytecodes. The analysis is *field-sensitive*, meaning that it tracks the individual fields of individual pointers. Our analysis is mostly flow-insensitive, using flow sensitivity only in the analysis of local pointers in each function. The results of this analysis, as we show in this paper, can be easily used to answer users' queries and to build more advanced analyses and programming tools.

1.1 Cloning to Achieve Context Sensitivity

Our approach to context sensitivity is based on the notion of *cloning*. Cloning conceptually generates multiple instances of a method such that every distinct calling context invokes a different instance, thus preventing information from one context to flow to another. Cloning makes generating context-sensitive results algorithmically trivial: We can simply apply a *context-insensitive* algorithm to the cloned program to obtain *context-sensitive* results. Note that our analysis does not clone the code per se; it simply produces a separate answer for each clone.

The context of a method invocation is often distinguished by its *call path*, which is simply the call sites, or return addresses, on the invocation's call stack. In the case of a recursive program, there are an unbounded number of calling contexts. To limit the number of calling contexts, Shivers proposed the concept of k-CFA (Control Flow Analysis) whereby one remembers only the last k call sites[26]. Emami et al. suggested distinguishing contexts by their full call paths if they are acyclic. For cyclic paths, they suggested including each call site in recursive cycles only once[14]. Our approach also uses entire call paths to distinguish between contexts in programs without recursion. To handle recursion, call paths are *reduced* by eliminating all invocations whose callers and callees belong to the same strongly connected component in the call graph. These reduced call paths are used to identify contexts.

It was not obvious, at least to us at the beginning of this project, that a cloning-based approach would be feasible. The number of reduced call paths in a program grows exponentially with the number of methods, and a cloning-based approach must compute the result of every one of these contexts. Emami et al. have only reported context-sensitive points-to results on small programs[14]. Realistic programs have many contexts; for example, the `megamek` application has over 10^{14} contexts (see Section 6.1). The size of the final results alone appears to be prohibitive.

We show that we can scale a cloning-based points-to analysis by representing the context-sensitive relations using ordered binary decision diagrams (BDDs)[6]. BDDs, originally designed for hardware verification, have previously been used in a number of program analyses[2, 23, 38], and more recently for points-to analysis[3, 39]. We show that it is possible to compute context-sensitive points-to results for over 10^{14} contexts.

In contrast, most context-sensitive pointer alias analyses developed to date are summary-based[15, 34, 37]. Parameterized summaries are created for each method and used in creating the summaries of its callers. It is not necessary to represent the results for the exponentially many contexts explicitly with this approach, because the result of a context can be computed independently using the summaries. However, to answer queries as simple as "which variables point to a certain object" would require all the results to be computed. The readers may be interested to know that, despite much effort, we tried but did not succeed in creating a scalable summary-based algorithm using BDDs.

1.2 Contributions

The contributions of this paper are not limited to just an algorithm for computing context-sensitive and inclusion-based points-to information. The methodology, specification language, representation, and tools we used in deriving our pointer analysis are applicable to creating many other algorithms. We demonstrate this by using the approach to create a variety of queries and algorithms.

Scalable cloning-based context-sensitive points-to analysis using BDDs. The algorithm we have developed is remarkably simple. We first create a cloned call graph where a clone is created for every distinct calling context. We then run a simple context-insensitive algorithm over the cloned call graph to get context-sensitive results. We handle the large number of contexts by representing them in BDDs and using an encoding scheme that allows commonalities among similar contexts to be exploited. We improve the efficiency of the algorithm by using an automatic tool that searches for an effective variable ordering.

Datalog as a high-level language for BDD-based program analyses. Instead of writing our program analyses directly in terms of BDD operations, we store all program information and results as relations and express our analyses in Datalog, a logic programming language used in deductive databases[30]. Because Datalog is succinct and declarative, we can express points-to analyses and many other algorithms simply and intuitively in just a few Datalog rules.

We use Datalog because its set-based operation semantics matches the semantics of BDD operations well. To aid our algorithm research, we have developed a deductive database system called `bddbddb` (BDD Based Deductive DataBase) that automatically translates Datalog programs into BDD algorithms. We provide a high-level summary of the optimizations in this paper; the details are beyond the scope of this paper[35].

Our experience is that programs generated by `bddbddb` are faster than their manually optimized counterparts. More importantly, Datalog programs are orders-of-magnitude easier to write. They are so succinct and easy to understand that we use them to explain all our algorithms here directly. All the experimental results reported in this paper are obtained by running the BDD programs automatically generated by `bddbddb`.

Context-sensitive queries and other analyses. The context-sensitive points-to results, the simple cloning-based approach to context sensitivity, and the `bddbddb` system make it easy to write new analyses. We show some representative examples in each of the following categories:

1. *Simple queries*. The results from our context-sensitive pointer analysis provide a wealth of information of interest to programmers. We show how a few lines of Datalog can help programmers debug a memory leak and find potential security vulnerabilities.
2. *Algorithms using context-sensitive points-to results*. We show how context-sensitive points-to results can be used to create advanced analyses. We include examples of a context-sensitive analysis to compute side effects (mod-ref) and an analysis to refine declared types of variables.
3. *Other context-sensitive algorithms*. Cloning can be used to trivially generate other kinds of context-sensitive results besides points-to relations. We illustrate this with a context-sensitive type analysis and a context-sensitive thread escape analysis. Whereas previous escape analyses require thousands of lines of code to implement[34], the algorithm here has only seven Datalog rules.

Experimental Results. We present the analysis time and memory usage of our analyses across 21 of the most popular Java applications on Sourceforge. Our context-sensitive pointer analysis can analyze even the largest of the programs in under 19 minutes. We also compare the precision of context-insensitive pointer analysis, context-sensitive pointer analysis and context-sensitive type analysis, and show the effects of merging versus cloning contexts.

1.3 Paper Organization

Here is an overview of the rest of the paper. Section 2 explains our methodology. Using Berndl's context-insensitive points-to algorithm as an example, we explain how an analysis can be expressed in Datalog and, briefly, how `bddbddb` translates Datalog into efficient BDD implementations. Section 3 shows how we can easily extend the basic points-to algorithm to discover call graphs on the fly by adding a few Datalog rules. Section 4 presents our cloning-based approach and how we use it to compute context-sensitive points-to results. Section 5 shows the representative queries and algorithms built upon our points-to results and the cloning-based approach. Section 6 presents our experimental results. We report related work in Section 7 and conclude in Section 8.

2. FROM DATALOG TO BDDS

In this section, we start with a brief introduction to Datalog. We then show how Datalog can be used to describe the context-insensitive points-to analysis due to Berndl et al. at a high level. We then describe how our `bddbddb` system translates a Datalog program into an efficient implementation using BDDs.

2.1 Datalog

We represent a program and all its analysis results as relations. Conceptually, a *relation* is a two-dimensional table. The columns are the *attributes*, each of which has a *domain* defining the set of possible attribute values. The rows are the tuples of attributes that share the relation. If tuple (x, y, z) is in relation A, we say that predicate $A(x, y, z)$ is true.

A Datalog program consists of a set of rules, written in a Prolog-style notation, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule

$$D(w,z) \quad :- \quad A(w,x), B(x,y), C(y,z).$$

says that "$D(w,z)$ is true if $A(w,x)$, $B(x,y)$, and $C(y,z)$ are all true." Variables in the predicates can be replaced with constants, which are surrounded by double-quotes, or don't-cares, which are signified by underscores. Predicates on the right side of the rules can be inverted.

Datalog is more powerful than SQL, which is based on relational calculus, because Datalog predicates can be recursively defined[30]. If none of the predicates in a Datalog program is inverted, then there is a guaranteed minimal solution consisting of relations with the least number of tuples. Conversely, programs with inverted predicates may not have a unique minimal solution. Our bddbddb system accepts a subclass of Datalog programs, known as *stratified* programs[7], for which minimal solutions always exist. Informally, rules in such programs can be grouped into strata, each with a unique minimal solution, that can be solved in sequence.

2.2 Context-Insensitive Points-to Analysis

We now review Berndl et al.'s context-insensitive points-to analysis[3], while also introducing the Datalog notation. This algorithm assumes that a call graph, computed using simple class hierarchy analysis[13], is available a priori. Heap objects are named by their allocation sites. The algorithm finds the objects possibly pointed to by each variable and field of heap objects in the program. Shown in Algorithm 1 is the exact Datalog program, as fed to bddbddb, that implements Berndl's algorithm. To keep the first example simple, we defer the discussion of using types to improve precision until Section 2.3.

ALGORITHM 1. *Context-insensitive points-to analysis with a precomputed call graph.*

DOMAINS

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

RELATIONS

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

RULES

$$vP(v,h) \quad :- \quad vP_0(v,h). \tag{1}$$

$$vP(v_1,h) \quad :- \quad assign(v_1,v_2), vP(v_2,h). \tag{2}$$

$$hP(h_1,f,h_2) \quad :- \quad store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \tag{3}$$

$$vP(v_2,h_2) \quad :- \quad load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2). \tag{4}$$

□

A Datalog program has three sections: domains, relations, and rules. A *domain declaration* has a name, a size n, and an optional file name that provides a name for each element in the domain, internally represented as an ordinal number from 0 to $n-1$. The latter allows bddbddb to communicate with the users with meaningful names. A *relation declaration* has an optional keyword specifying whether it is an input or output relation, the name of the relation, and the name and domain of every attribute. A relation declared as neither input nor output is a temporary relation generated in the analysis but not written out. Finally, the rules follow the standard Datalog syntax. The rule numbers, introduced here for the sake of exposition, are not in the actual program.

We can express all information found in the intermediate representation of a program as relations. To avoid inundating readers with too many definitions all at once, we define the relations as they are used. The domains and relations used in Algorithm 1 are:

V is the domain of variables. It represents all the allocation sites, formal parameters, return values, thrown exceptions, cast operations, and dereferences in the program. There is also a special *global* variable for use in accessing static variables.

H is the domain of heap objects. Heap objects are named by the invocation sites of object creation methods. To increase precision, we also statically identify factory methods and treat them as object creation methods.

F is the domain of field descriptors in the program. Field descriptors are used when loading from a field (`v2 = v1.f;`) or storing to a field (`v1.f = v2;`). There is a special field descriptor to denote an array access.

vP_0: V × H is the initial variable points-to relation extracted from object allocation statements in the source program. $vP_0(v,h)$ means there is an invocation site h that assigns a newly allocated object to variable v.

$store$: V × F × V represents store statements. $store(\mathtt{v1},\mathtt{f},\mathtt{v2})$ says that there is a statement "`v1.f = v2;`" in the program.

$load$: V × F × V represents load statements. $load(\mathtt{v1},\mathtt{f},\mathtt{v2})$ says that there is a statement "`v2 = v1.f;`" in the program.

$assign$: V × V is the assignments relation due to passing of arguments and return values. $assign(v_1,v_2)$ means that variable v_1 includes the points-to set of variable v_2. Although we do not cover exceptions here, they work in an analogous manner.

vP: V × H is the output variable points-to relation. $vP(v,h)$ means that variable v can point to heap object h.

hP: H × F × H is the output heap points-to relation. $hP(h_1,f,h_2)$ means that field f of heap object h_1 can point to heap object h_2.

Note that local variables and their assignments are factored away using a flow-sensitive analysis[33]. The *assign* relation is derived by using a precomputed call graph. The sizes of the domains are determined by the number of variables, heap objects, and field descriptors in the input program.

Rule (1) incorporates the initial variable points-to relations into vP. Rule (2) finds the transitive closure over inclusion edges. If v_1 includes v_2 and variable v_2 can point to object h, then v_1 can also point to h. Rule (3) models the effect of store instructions on heap objects. Given a statement "`v1.f = v2;`", if `v1` can point to h_1 and `v2` can point to h_2, then h_1.`f` can point to h_2. Rule (4) resolves load instructions. Given a statement "`v2 = v1.f;`", if `v1` can point to h_1 and h_1.`f` can point to h_2, then `v2` can point to h_2. Applying these rules until the results converge finds all the possible context-insensitive points-to relations in the program.

2.3 Improving Points-to Analysis with Types

Because Java is type-safe, variables can only point to objects of *assignable* types. *Assignability* is similar to the subtype relation,

ALGORITHM 2. *Context-insensitive points-to analysis with type filtering.*

DOMAINS

Domains from Algorithm 1, plus:

T	4096	`type.map`

RELATIONS

Relations from Algorithm 1, plus:

input	vT	($variable$: V, $type$: T)
input	hT	($heap$: H, $type$: T)
input	aT	($supertype$: T, $subtype$: T)
	$vPfilter$	($variable$: V, $heap$: H)

RULES

$$vPfilter(v,h) :- vT(v,t_v), hT(h,t_h), aT(t_v,t_h). \quad (5)$$

$$vP(v,h) :- vP_0(v,h). \quad (6)$$

$$vP(v_1,h) :- assign(v_1,v_2), vP(v_2,h), vPfilter(v_1,h). \quad (7)$$

$$hP(h_1,f,h_2) :- store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \quad (8)$$

$$vP(v_2,h_2) :- load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2), vPfilter(v_2,h_2). \quad (9)$$

□

with allowances for interfaces, null values, and arrays[22]. By dropping targets of unassignable types in assignments and load statements, we can eliminate many impossible points-to relations that result from the imprecision of the analysis. [1]

Adding type filtering to Algorithm 1 is simple in Datalog. We add a new domain to represent types and new relations to represent assignability as well as type declarations of variables and heap objects. We compute the type filter and modify the rules in Algorithm 1 to filter out unsafe assignments and load operations.

T is the domain of type descriptors (i.e. classes) in the program.

vT: V × T represents the declared types of variables. $vT(v,t)$ means that variable v is declared with type t.

hT: H × T represents the types of objects created at a particular creation site. In Java, the type created by a `new` instruction is usually known statically.[2] $hT(h,t)$ means that the object created at h has type t.

aT: T × T is the relation of assignable types. $aT(t_1,t_2)$ means that type t_2 is assignable to type t_1.

$vPfilter$: V × H is the type filter relation. $vPfilter(v,h)$ means that it is type-safe to assign heap object h to variable v.

Rule (5) in Algorithm 2 defines the $vPfilter$ relation: It is type-safe to assign heap object h of type t_h to variable v of type t_v if t_v is assignable from t_h. Rules (6) and (8) are the same as Rules (1) and (3) in Algorithm 1. Rules (7) and (9) are analogous to Rules (2) and (4), with the additional constraint that only points-to relations that match the type filter are inserted.

[1]We could similarly perform type filtering on stores into heap objects. However, because all stores must go through variables, such a type filter would only catch one extra case — when the base object is a null constant.

[2]The type of a created object may not be known precisely if, for example, the object is returned by a native method or reflection is used. Such types are modeled conservatively as all possible types.

2.4 Translating Datalog into Efficient BDD Implementations

We first describe how Datalog rules can be translated into operators from relational algebra such as "join" and "project", then show how to translate these operations into BDD operations.

2.4.1 Query Resolution

We can find the solution to an unstratified query, or a stratum of a stratified query, simply by applying the inference rules repeatedly until none of the output relations change. We can apply a Datalog rule by performing a series of relational natural join, project and rename operations. A natural join operation combines rows from two relations if the rows share the same value for a common attribute. A project operation removes an attribute from a relation. A rename operation changes the name of an attribute to another one.

For example, the application of Rule (2) can be implemented as:

$$\begin{aligned} t_1 &= \text{rename}(vP, variable, source);\\ t_2 &= \text{project}(\text{join}(assign, t_1), source);\\ vP &= vP \cup \text{rename}(t_2, dest, variable); \end{aligned}$$

We first rename the attribute in relation vP from *variable* to *source* so that it can be joined with relation *assign* to create a new points-to relation. The attribute *dest* of the resulting relation is changed to *variable* so that the tuples can be added to the vP tuples accumulated thus far.

The bddbddb system uses the three following optimizations to speed up query resolution.

Attributes naming. Since the names of the attributes must match when two relations are joined, the choice of attribute names can affect the costs of rename operations. Since the renaming cost is highly sensitive to how the relations are implemented, the bddbddb system takes the representation into account when minimizing the renaming cost.

Rule application order. A rule needs to be applied only if the input relations have changed. bddbddb optimizes the ordering of the rules by analyzing the dependences between the rules. For example, Rule 1 in Algorithm 1 does not depend on any of the other rules and can be applied only once at the beginning of the query resolution.

Incrementalization. We only need to re-apply a rule on those combinations of tuples that we have not seen before. Such a technique is known as incrementalization in the BDD literature and semi-naïve fixpoint evaluation in the database literature[1]. Our system also identifies loop-invariant relations to avoid unnecessary difference and rename operations. Shown below is the result of incrementalizing the repeated application of Rule (2):

$$\begin{aligned} &d = vP;\\ &\text{repeat}\\ &\quad t_1 = \text{rename}(d, variable, source);\\ &\quad t_2 = \text{project}(\text{join}(assign, t_1), source);\\ &\quad d' = \text{rename}(t_2, dest, variable);\\ &\quad d = d' - vP;\\ &\quad vP = vP \cup d;\\ &\text{until } d == \emptyset; \end{aligned}$$

2.4.2 Relational Algebra in BDD

We now explain how BDDs work and how they can be used to implement relations and relational operations. BDDs (Binary Decision Diagrams) were originally invented for hardware verification to efficiently store a large number of states that share many commonalities[6]. They are an efficient representation of boolean functions.

A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes, representing the constants one and zero. Each non-terminal node in the DAG represents an input variable and has exactly two outgoing edges: a high edge and a low edge. The high edge represents the case where the input variable for the node is true, and the low outgoing edge represents the case where the input variable is false. On any path in the DAG from the root to a terminal node, the value of the function on the truth values on the input variables in the path is given by the value of the terminal node. To evaluate a BDD for a specific input, one simply starts at the root node and, for each node, follows the high edge if the input variable is true, and the low edge if the input variable is false. The value of the terminal node that we reach is the value of the BDD for that input.

The variant of BDDs that we use are called *ordered binary decision diagrams*, or OBDDs[6]. "Ordered" refers to the constraint that on all paths through the graph the variables respect a given linear order. In addition, OBDDs are *maximally reduced* meaning that nodes with the same variable name and low and high successors are collapsed as one, and nodes with identical low and high successors are bypassed. Thus, the more commonalities there are in the paths leading to the terminals, the more compact the OBDDs are. Accordingly, the amount of the sharing and the size of the representation depends greatly on the ordering of the variables.

We can use BDDs to represent relations as follows. Each element d in an n-element domain D is represented as an integer between 0 and $n-1$ using $log_2(n)$ bits. A relation $R : D_1 \times \ldots \times D_n$ is represented as a boolean function $f : D_1 \times \ldots \times D_n \rightarrow \{0,1\}$ 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 $f(d_1, \ldots, d_n) = 0$.

A number of highly-optimized BDD packages are available[21, 27]; the operations they provide can be used directly to implement relational operations efficiently. For example, the "replace" operation in BDD has the same semantics as the "rename" operation; the "relprod" operation in BDD finds the natural join between two relations and projects away the common domains.

Let us now use a concrete example to illustrate the significance of variable ordering. Suppose relation R_1 contains tuples $(1,1),(2,1),\ldots,(100,1)$ and relation R_2 contains tuples $(1,2),(2,2),\ldots,(100,2)$. If in the variable order the bits for the first attribute come before the bits for the second, the BDD will need to represent the sequence $1,\ldots,100$ separately for each relation. However, if instead the bits for the second attribute come first, the BDD can share the representation for the sequence $1,\ldots,100$ between R_1 and R_2. Unfortunately, the problem of finding the best variable ordering is NP-complete[5]. Our bddbddb system automatically explores different alternatives empirically to find an effective ordering[35].

3. CALL GRAPH DISCOVERY

The call graph generated using class hierarchy analysis can have many spurious call targets, which can lead to many spurious points-to relations[19]. We can get more precise results by creating the call graph on the fly using points-to relations. As the algorithm generates points-to results, they are used to identify the receiver types of the methods invoked and to bind calls to target methods; and as call graph edges are discovered, we use them to find more points-to relations. The algorithm converges when no new call targets and no new pointer relations are found.

Modifying Algorithm 2 to discover call graphs on the fly is simple. Instead of an input $assign$ relation computed from a given call graph, we derive it from method invocation statements and points-to relations.

ALGORITHM 3. *Context-insensitive points-to analysis that computes call graph on the fly.*

DOMAINS

Domains from Algorithm 2, plus:

I	32768	`invoke.map`
N	4096	`name.map`
M	16384	`method.map`
Z	256	

RELATIONS

Relations from Algorithm 2, with the modification that $assign$ is now a computed relation, plus:

input	cha	$(type : \mathrm{T}, name : \mathrm{N}, target : \mathrm{M})$
input	$actual$	$(invoke : \mathrm{I}, param : \mathrm{Z}, var : \mathrm{V})$
input	$formal$	$(method : \mathrm{M}, param : \mathrm{Z}, var : \mathrm{V})$
input	IE_0	$(invoke : \mathrm{I}, target : \mathrm{M})$
input	mI	$(method : \mathrm{M}, invoke : \mathrm{I})$
output	IE	$(invoke : \mathrm{I}, target : \mathrm{M})$

RULES

Rules from Algorithm 2, plus:

$$IE(i,m) \quad :- \quad IE_0(i,m). \tag{10}$$

$$IE(i,m_2) \quad :- \quad mI(m_1,i,n), actual(i,0,v), vP(v,h), hT(h,t), cha(t,n,m_2). \tag{11}$$

$$assign(v_1,v_2) \quad :- \quad IE(i,m), formal(m,z,v_1), actual(i,z,v_2). \tag{12}$$

□

I is the domain of invocation sites in the program. An invocation site is a method invocation of the form `r = p0.m(p1 ... pk)`. Note that $H \subseteq I$.

N is the domain of method names used in invocations. In an invocation `r = p0.n(p1 ... pk)`, `n` is the method name.

M is the domain of implemented methods in the program. It does not include abstract or interface methods.

Z is the domain used for numbering parameters.

cha: T × N × M encodes virtual method dispatch information from the class hierarchy. $cha(t,n,m)$ means that m is the target of dispatching the method name n on type t.

$actual$: I × Z × V encodes the actual parameters for invocation sites. $actual(i,z,v)$ means that v is passed as parameter number z at invocation site i.

$formal$: M × Z × V encodes formal parameters for methods. $formal(m,z,v)$ means that formal parameter z of method m is represented by variable v.

IE_0: I × M are the initial invocation edges. They record the invocation edges whose targets are statically bound. In Java, some calls are static or non-virtual. Additionally, local type analysis combined with analysis of the class hierarchy allows us to determine that some calls have a single target[13]. $IE_0(i,m)$ means that invocation site i can be analyzed statically to call method m.

mI: M × I × N represents invocation sites. $mI(m,i,n)$ means that method m contains an invocation site i with virtual method name n. Non-virtual invocation sites are given a special null method name, which does not appear in the cha relation.

IE: I × M is an output relation encoding all invocation edges. $IE(i,m)$ means that invocation site i calls method m.

The rules in Algorithm 3 compute the $assign$ relation used in Algorithm 2. Rules (10) and (11) find the invocation edges, with the former handling statically bound targets and the latter handling virtual calls. Rule (11) matches invocation sites with the type of the "this" pointer and the class hierarchy information to find the possible target methods. If an invocation site i with method name n is invoked on variable v, and v can point to h and h has type t, and invoking n on type t leads to method m, then m is a possible target of invocation i.

Rule (12) handles parameter passing.[3] If invocation site i has a target method m, variable v_2 is passed as argument number z, and the formal parameter z of method m is v_1, then the points-to set of v_1 includes the points-to set of v_2. Return values are handled in a likewise manner, only the inclusion relation is in the opposite direction. We see that as the discovery of more variable points-to (vP) can create more invocation edges (IE), which in turn can create more assignments ($assign$) and more points-to relations. The algorithm converges when all the relations stabilize.

4. CONTEXT SENSITIVE POINTS-TO

A context-insensitive or *monomorphic* analysis produces just one set of results for each method regardless how many ways a method may be invoked. This leads to imprecision because information from different calling contexts must be merged, so information along one calling context can propagate to other calling contexts. A context-sensitive or *polymorphic* analysis avoids this imprecision by allowing different contexts to have different results.

We can make a context-sensitive version of a context-insensitive analysis as follows. We make a clone of a method for each path through the call graph, linking each call site to its own unique clone. We then run the original context-insensitive analysis over the exploded call graph. However, this technique can require an exponential (and in the presence of cycles, potentially unbounded) number of clones to be created.

It has been observed that different contexts of the same method often have many similarities. For example, parameters to the same method often have the same types or similar aliases. This observation led to the concept of partial transfer functions (PTF), where summaries for each input pattern are created on the fly as they are discovered[36, 37]. However, PTFs are notoriously difficult to implement and get correct, as the programmer must explicitly calculate the input patterns and manage the summaries. Furthermore, the technique has not been shown to scale to very large programs.

Our approach is to allow the exponential explosion to occur and rely on the underlying BDD representation to find and exploit the commonalities across contexts. BDDs can express large sets of redundant data in an efficient manner. Contexts with identical information will automatically be shared at the data structure level. Furthermore, because BDDs operate down at the bit level, it can even exploit commonalities between contexts with different information. BDD operations operate on entire relations at a time, rather than one tuple at a time. Thus, the cost of BDD operations depends on the size and shape of the BDD relations, which depends greatly on the variable ordering, rather than the number of tuples in a relation. Also, due to caching in BDD packages, identical subproblems only have to be computed once. Thus, with the right variable ordering, the results for all contexts can be computed very efficiently.

4.1 Numbering Call Paths

A *call path* is a sequence of invocation edges $(i_1, m_1), (i_2, m_2), \ldots,$ such that i_1 is an invocation site in an entry method, typically `main`[4], and i_k is an invocation site in method m_{k-1} for all $k > 1$.

[3]We also match thread objects to their corresponding `run()` methods, even though the edges do not explicitly appear in the call graph.

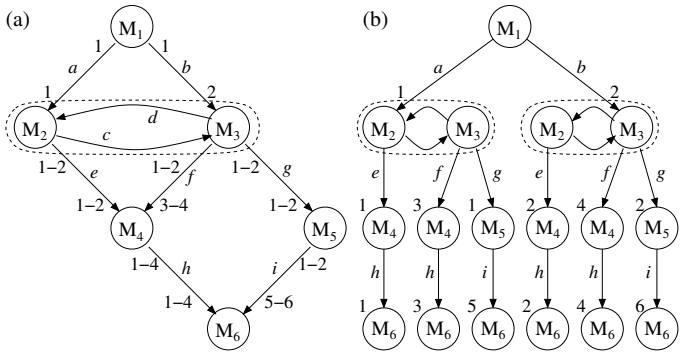

Figure 1: Example of path numbering. The graph on the left is the original graph. Nodes M_2 and M_3 are in a cycle and therefore are placed in one equivalence class. Each edge is marked with path numbers at the source and target of the edge. The graph on the right is the graph with all of the paths expanded.

Call paths reaching M_6	Reduced call paths reaching M_6
$a(cd)^*eh$	aeh
$b(dc)^*deh$	beh
$a(cd)^*cfh$	afh
$b(dc)^*fh$	bfh
$a(cd)^*cgi$	agi
$b(dc)^*gi$	bgi

Figure 2: The six contexts of function M_6 in Example 1

For programs without recursion, every call path to a method defines a context for that method. To handle recursive programs, which have an unbounded number of call paths, we first find the strongly connected components (SCCs) in a call graph. By eliminating all method invocations whose caller and callee belong to the same SCC from the call paths, we get a finite set of *reduced call paths*. Each reduced call path to an SCC defines a context for the methods in the SCC. Thus, information from different paths leading to the SCCs are kept separate, but the methods within the SCC invoked with the same incoming call path are analyzed context-insensitively.

EXAMPLE 1. Figure 1(a) shows a small call graph with just six methods and a set of invocation edges. Each invocation edge has a name, being one of a through i; its source is labeled by the context number of the caller and its sink by the context number of the callee. The numbers will be explained in Example 2. Methods M_2 and M_3 belong to a strongly connected component, so invocations along edges c and d are eliminated in the computation of reduced call graphs. While there are infinitely many call paths reaching method M_6, there are only six reduced call paths reaching M_6, as shown in Figure 2. Thus M_6 has six clones, one for each reduced call path.

Under this definition of context sensitivity, large programs can have many contexts. For example, `pmd` from our test programs has 1971 methods and 10^{23} contexts! In the BDD representation, we give each reduced call path reaching a method a distinct *context*

[4]Other "entry" methods in typical programs are static class initializers, object finalizers, and thread run methods.

number. It is important to find a context numbering scheme that allows the BDDs to share commonalities across contexts. Algorithm 4 shows one such scheme.

ALGORITHM 4. *Generating context-sensitive invocation edges from a call graph.*

INPUT: A call multigraph.

OUTPUT: Context-sensitive invocation edges IE_c: C × I × C × M, where C is the domain of context numbers. $IE_c(c, i, c_m, m)$ means that invocation site i in context c calls method m in context c_m.

METHOD:

1. A method with n clones will be given numbers $1, \ldots, n$. Nodes with no predecessors are given a singleton context numbered 1.
2. Find strongly connected components in the input call graph. The ith clone of a method always calls the ith clone of another method belonging to the same component.
3. Collapse all methods in a strongly connected component to a single node to get an acyclic reduced graph.
4. For each node n in the reduced graph in topological order,
   Set the counts of contexts created, c, to 0.
   For each incoming edge,
   If the predecessor of the edge p has k contexts,
   create k clones of node n,
   Add tuple $(i, p, i + c, n)$ to IE_c, for $1 \leq i \leq k$,
   $c = c + k$.

□

EXAMPLE 2. We now show the results of applying Algorithm 4 to Example 1. M_1, the root node, is given context number 1. We shall visit the invocation edges from left to right. Nodes M_2 and M_3, being members of a strongly connected component, are represented as one node. The strongly connected component is reached by two edges from M_1. Since M_1 has only one context, we create two clones, one reached by each edge. For method M_4, the predecessor on each of the two incoming edges has two contexts, thus M_4 has four clones. Method M_5 has two clones, one for each clone that invokes M_5. Finally, method M_6 has six clones: Clones 1-4 of method M_4 invoke clones 1-4 and clones 1-2 of method M_5 call clones 5-6, respectively. The cloned graph is shown in Figure 1(b).

The numbering scheme used in Algorithm 4 plays up the strengths of BDDs. Each method is assigned a contiguous range of contexts, which can be represented efficiently in BDDs. The contexts of callees can be computed simply by adding a constant to the contexts of the callers; this operation is also cheap in BDDs. Because the information for contexts that share common tail sequences are likely to be similar, this numbering allows the BDD data structure to share effectively across common contexts. For example, the sequentially-numbered clones 1 and 2 of M_6 both have a common tail sequence eh. Because of this, the contexts are likely to be similar and therefore the BDD can take advantage of the redundancies.

To optimize the creation of the cloned invocation graph, we have defined a new primitive that creates a BDD representation of contiguous ranges of numbers in O(k) operations, where k is the number of bits in the domain. In essence, the algorithm creates one BDD to represent numbers below the upper bound, and one to represent numbers above the lower bound, and computes the conjunction of these two BDDs.

4.2 Context-Sensitive Pointer Analysis with a Pre-computed Call Graph

We are now ready to present our context-sensitive pointer analysis. We assume the presence of a pre-computed call graph created, for example, by using a context-insensitive points-to analysis (Algorithm 3). We apply Algorithm 4 to the call graph to generate the context-sensitive invocation edges IE_c. Once that is created, we can simply apply a context-insensitive points-to analysis on the exploded call graph to get context-sensitive results. We keep the results separate for each clone by adding a context number to methods, variables, invocation sites, points-to relations, etc.

ALGORITHM 5. *Context-sensitive points-to analysis with a pre-computed call graph.*

DOMAINS

Domains from Algorithm 2, plus:

C	9223372036854775808

RELATIONS

Relations from Algorithm 2, plus:

input	IE_c	($caller$: C, $invoke$: I, $callee$: C, tgt : M)
	$assign_c$	($dest_c$: C, $dest$: V, src_c : C, src : V)
output	vP_c	($context$: C, $variable$: V, $heap$: H)

RULES

$$vPfilter(v, h) :- vT(v, t_v), hT(h, t_h), aT(t_v, t_h). \quad (13)$$

$$vP_c(c, v, h) :- vP_0(v, h), IE_c(c, h, \_, \_). \quad (14)$$

$$vP_c(c_1, v_1, h) :- assign_c(c_1, v_1, c_2, v_2), vP_c(c_2, v_2, h), vPfilter(v_1, h). \quad (15)$$

$$hP(h_1, f, h_2) :- store(v_1, f, v_2), vP_c(c, v_1, h_1), vP_c(c, v_2, h_2). \quad (16)$$

$$vP_c(c, v_2, h_2) :- load(v_1, f, v_2), vP_c(c, v_1, h_1), hP(h_1, f, h_2), vPfilter(v_2, h_2). \quad (17)$$

$$assign_c(c_1, v_1, c_2, v_2) :- IE_c(c_2, i, c_1, m), formal(m, z, v_1), actual(i, z, v_2). \quad (18)$$

□

C is the domain of context numbers. Our BDD library uses signed 64-bit integers to represent domains, so the size is limited to 2^{63}.

IE_c: C × I × C × M is the set of context-sensitive invocation edges. $IE_c(c, i, c_m, m)$ means that invocation site i in context c calls method m in context c_m. This relation is computed using Algorithm 4.

$assign_c$: C × V × C × V is the context-sensitive version of the $assign$ relation. $assign_c(c_1, v_1, c_2, v_2)$ means variable v_1 in context c_1 includes the points-to set of variable v_2 in context v_2 due to parameter passing. Again, return values are handled analogously.

vP_c: C × V × H is the context-sensitive version of the variable points-to relation (vP). $vP_c(c, v, h)$ means variable v in context c can point to heap object h.

Rule (18) interprets the context-sensitive invocation edges to find the bindings between actual and formal parameters. The rest of the rules are the context-sensitive counterparts to those found in Algorithm 2.

Algorithm 5 takes advantage of a pre-computed call graph to create an efficient context numbering scheme for the contexts. We can

compute the call graph on the fly while enjoying the benefit of the numbering scheme by numbering all the *possible contexts* with a conservative call graph, and delaying the generation of the invocation edges only if warranted by the points-to results. We can reduce the iterations necessary by exploiting the fact that many of the invocation sites of a call graph created by a context-insensitive analysis have single targets. Such an algorithm has an execution time similar to Algorithm 5, but is of primarily academic interest as the call graph rarely improves due to the extra precision from context-sensitive points-to information.

5. QUERIES AND OTHER ANALYSES

The algorithms in sections 2, 3 and 4 generate vast amounts of results in the form of relations. Using the same declarative programming interface, we can conveniently query the results and extract exactly the information we are interested in. This section shows a variety of queries and analyses that make use of pointer information and context sensitivity.

5.1 Debugging a Memory Leak

Memory leaks can occur in Java when a reference to an object remains even after it will no longer be used. One common approach of debugging memory leaks is to use a dynamic tool that locates the allocation sites of memory-consuming objects. Suppose that, upon reviewing the information, the programmer thinks objects allocated in line 57 in file `a.java` should have been freed. He may wish to know which objects may be holding pointers to the leaked objects, and which operations may have stored the pointers. He can consult the static analysis results by supplying the queries:

$$\begin{array}{lll} whoPointsTo57(h,f) & :- & hP(h,f,\text{“}\texttt{a.java}:57\text{”}). \\ whoDunnit(c,v_1,f,v_2) & :- & store(v_1,f,v_2), \\ & & vP_{\text{C}}(c,v_2,\text{“}\texttt{a.java}:57\text{”}). \end{array}$$

The first query finds the objects and their fields that may point to objects allocated at "`a.java`:57"; the second finds the store instructions, and the contexts under which they are executed, that create the references.

5.2 Finding a Security Vulnerability

The Java Cryptography Extension (JCE) is a library of cryptographic algorithms[29]. Misuse of the JCE API can lead to security vulnerabilities and a false sense of security. For example, many operations in the JCE use a secret key that must be supplied by the programmer. It is important that secret keys be cleared after they are used so they cannot be recovered by attackers with access to memory. Since `String` objects are immutable and cannot be cleared, secret keys should not be stored in `String` objects but in an array of characters or bytes instead.

To guard against misuse, the function that accepts the secret key, `PBEKeySpec.init()`, only allows arrays of characters or bytes as input. However, a programmer not versed in security issues may have stored the key in a `String` object and then use a routine in the `String` class to convert it to an array of characters. We can write a query to audit programs for the presence of such idioms. Let $Mret(m,v)$ be an input relation specifying that variable v is the return value of method m. We define a relation $fromString(h)$ which indicates if the object h was directly derived from a `String`. Specifically, it records the objects that are returned by a call to a method in the `String` class. An invocation i to method `PBEKeySpec.init()` is a vulnerability if the first argument points to an object derived from a `String`.

$$\begin{array}{lll} fromString(h) & :- & cha(\text{“}\texttt{String}\text{”},\_,m), Mret(m,v), \\ & & vP_{\text{C}}(\_,v,h). \end{array}$$

$$\begin{array}{lll} vuln(c,i) & :- & IE(i,\text{“}\texttt{PBEKeySpec.init()}\text{”}), \\ & & actual(i,1,v), vP_{\text{C}}(c,v,h), \\ & & fromString(h). \end{array}$$

Notice that this query does not only find cases where the object derived from a `String` is immediately supplied to `PBEKeySpec.init()`. This query will also identify cases where the object has passed through many variables and heap objects.

5.3 Type Refinement

Libraries are written to handle the most general types of objects possible, and their full generality is typically not used in many applications. By analyzing the actual types of objects used in an application, we can *refine* the types of the variables and object fields. Type refinement can be used to reduce overheads in cast operations, resolve virtual method calls, and gain better understanding of the program.

We say that variable v can be legally declared as t, written $varSuperTypes(v,t)$, if t is a supertype of the types of all the objects v can point to. The type of a variable is refinable if the variable can be declared to have a more precise type. To compute the super types of v, we first find $varExactTypes(v,t)$, the types of objects pointed to by v. We then intersect the supertypes of all the exact types to get the desired solution; we do so in Datalog by finding the complement of the union of the complement of the exact types.

$$\begin{array}{lll} varExactTypes(v,t) & :- & vP_{\text{C}}(\_,v,h), hT(h,t). \\ notVarType(v,t) & :- & varExactTypes(v,t_v), \neg aT(t,t_v). \\ varSuperTypes(v,t) & :- & \neg notVarType(v,t). \\ refinable(v,t_c) & :- & vT(v,t_d), varSuperTypes(v,t_c), \\ & & aT(t_d,t_c), t_d \neq t_c. \end{array}$$

The above shows a context-insensitive type refinement query. We find, for each variable, the type to which it can be refined regardless of the context. Even if the end result is context-insensitive, it is more precise to take advantage of the context-sensitive points-to results available to determine the exact types, as shown in the first rule. In Section 6.3, we compare the accuracy of this context-insensitive query with a context-sensitive version.

5.4 Context-Sensitive Mod-Ref Analysis

Mod-ref analysis is used to determine what fields of what objects may be modified or referenced by a statement or call site[18]. We can use the context-sensitive points-to results to solve a context-sensitive version of this query. We define $mV(m,v)$ to mean that v is a local variable in m. The mV^*_{C} relation specifies the set of variables and contexts of methods that are transitively reachable from a method. $mV^*_{\text{C}}(c_1,m,c_2,v)$ means that calling method m with context c_1 can transitively call a method with local variable v under context c_2.

$$\begin{array}{lll} mV^*_{\text{C}}(c,m,c,v) & :- & mV(m,v). \\ mV^*_{\text{C}}(c_1,m_1,c_3,v_3) & :- & mI(m_1,i), IE_{\text{C}}(c_1,i,c_2,m_2), \\ & & mV^*_{\text{C}}(c_2,m_2,c_3,v_3). \end{array}$$

The first rule simply says that a method m in context c can reach its local variable. The second rule says that if method m_1 in context c_1 calls method m_2 in context c_2, then m_1 in context c_1 can also reach all variables reached by method m_2 in context c_2.

We can now define the mod and ref set of a method as follows:

$$\begin{array}{lll} mod(c,m,h,f) & :- & mV^*_{\text{C}}(c,m,c_v,v), \\ & & store(v,f,\_), vP_{\text{C}}(c_v,v,h). \\ ref(c,m,h,f) & :- & mV^*_{\text{C}}(c,m,c_v,v), \\ & & load(v,f,\_), vP_{\text{C}}(c_v,v,h). \end{array}$$

The first rule says that if method m in context c can reach a variable v in context c_v, and if there is a store through that variable to field f of object h, then m in context c can modify field f of object h. The second rule for defining the ref relations is analogous.

5.5 Context-Sensitive Type Analysis

Our cloning technique can be applied to add context sensitivity to other context-insensitive algorithms. The example we show here is the type inference of variables and fields. By not distinguishing between instances of heap objects, this analysis does not generate results as precise as those extracted from running the complete context-sensitive pointer analysis as discussed in Section 5.3, but is much faster.

The basic type analysis is similar to 0-CFA[26]. Each variable and field in the program has a set of concrete types that it can refer to. The sets are propagated through calls, returns, loads, and stores. By using the path numbering scheme in Algorithm 4, we can convert this basic analysis into one which is context-sensitive—in essence, making the analysis into a k-CFA analysis where k is the depth of the call graph and recursive cycles are collapsed.

ALGORITHM 6. *Context-sensitive type analysis.*

DOMAINS

Domains from Algorithm 5

RELATIONS

Relations from Algorithm 5, plus:

output	vT_{C}	($context$: C, $variable$: V, $type$: T)
output	fT	($field$: F, $target$: T)
	$vTfilter$	($variable$: V, $type$: T)

RULES

$$vTfilter(v,t) \text{ :- } vT(v,t_v), aT(t_v,t). \quad (19)$$

$$vT_{\mathrm{C}}(c,v,t) \text{ :- } vP_0(v,h), IE_{\mathrm{C}}(c,h,\_,\_), hT(h,t). \quad (20)$$

$$vT_{\mathrm{C}}(c_{v_1},v_1,t) \text{ :- } assign_{\mathrm{C}}(c_{v_1},v_1,c_{v_2},v_2), vT_{\mathrm{C}}(c_{v_2},v_2,t), vTfilter(v_1,t). \quad (21)$$

$$fT(f,t) \text{ :- } store(\_,f,v_2), vT_{\mathrm{C}}(\_,v_2,t). \quad (22)$$

$$vT_{\mathrm{C}}(\_,v,t) \text{ :- } load(\_,f,v), fT(f,t), vTfilter(v,t). \quad (23)$$

$$assign_{\mathrm{C}}(c_1,v_1,c_2,v_2) \text{ :- } IE_{\mathrm{C}}(c_2,i,c_1,m), formal(m,z,v_1), actual(i,z,v_2). \quad (24)$$

□

vT_{C}: C × V × T is the context-sensitive variable type relation. $vT_{\mathrm{C}}(c,v,t)$ means that variable v in context c_v can refer to an object of type t. This is the analogue of vP_{C} in the points-to analysis.

fT: F × T is the field type relation. $fT(f,t)$ means that field f can point to an object of type t.

$vTfilter$: V × T is the type filter relation. $vTfilter(v,t)$ means that it is type-safe to assign an object of type t to variable v.

Rule (20) initializes the vT_{C} relation based on the initial local points-to information contained in vP_0, combining it with hT to get the type and IE_{C} to get the context numbers. Rule (21) does transitive closure on the vT_{C} relation, filtering with $vTfilter$ to enforce type safety. Rules (22) and (23) handle stores and loads, respectively. They differ from their counterparts in the pointer analysis in that they do not use the base object, only the field. Rule (24) models the effects of parameter passing in a context-sensitive manner.

5.6 Thread Escape Analysis

Our last example is a thread escape analysis, which determines if objects created by one thread may be used by another. The results of the analysis can be used for optimizations such as synchronization elimination and allocating objects in thread-local heaps, as well as for understanding programs and checking for possible race conditions due to missing synchronizations[8, 34]. This example illustrates how we can vary context sensitivity to fit the needs of the analysis.

We say that an object allocated by a thread has *escaped* if it may be *accessed* by another thread. This notion is stronger than most other formulations where an object is said to escape if it can be *reached* by another thread[8, 34].

Java threads, being subclasses of `java.lang.Thread`, are identified by their creation sites. In the special case where a thread creation can execute only once, a thread can simply be named by the creation site. The thread that exists at virtual machine startup is an example of a thread that can only be created once. A creation site reached via different call paths or embedded in loops or recursive cycles may generate multiple threads. To distinguish between thread instances created at the same site, we create two thread contexts to represent two separate thread instances. If an object created by one instance is not accessed by its clone, then it is not accessed by any other instances created by the same call site. This scheme creates at most twice as many contexts as there are thread creation sites.

We clone the thread `run()` method, one for each thread context, and place these clones on the list of entry methods to be analyzed. Methods (transitively) invoked by a context's `run()` method all inherit the same context. A clone of a method not only has its own cloned variables, but also its own cloned object creation sites. In this way, objects created by separate threads are distinct from each other. We run a points-to analysis over this slightly expanded call graph; an object created in a thread context escapes if it is accessed by variables in another thread context.

ALGORITHM 7. *Thread-sensitive pointer analysis.*

DOMAINS

Domains from Algorithm 5

RELATIONS

Relations from Algorithm 2, plus:

input	H_{T}	(c : C, $heap$: H)
input	$vP_{0\mathrm{T}}$	(cv : C, $variable$: V, ch : C, $heap$: H)
output	vP_{T}	(cv : C, $variable$: V, ch : C, $heap$: H)
output	hP_{T}	(cb : C, $base$: H, $field$: F, ct : C, $target$: H)

RULES

$$vPfilter(v,h) \text{ :- } vT(v,t_v), hT(h,t_h), aT(t_v,t_h). \quad (25)$$

$$vP_{\mathrm{T}}(c_1,v,c_2,h) \text{ :- } vP_{0\mathrm{T}}(c_1,v,c_2,h). \quad (26)$$

$$vP_{\mathrm{T}}(c,v,c,h) \text{ :- } vP_0(v,h), H_{\mathrm{T}}(c,h). \quad (27)$$

$$vP_{\mathrm{T}}(c_2,v_1,c_h,h) \text{ :- } assign(v_1,v_2), vP_{\mathrm{T}}(c_2,v_2,c_h,h), vPfilter(v_1,h). \quad (28)$$

$$hP_{\mathrm{T}}(c_1,h_1,f,c_2,h_2) \text{ :- } store(v_1,f,v_2), vP_{\mathrm{T}}(c,v_1,c_1,h_1), vP_{\mathrm{T}}(c,v_2,c_2,h_2). \quad (29)$$

$$vP_{\mathrm{T}}(c,v_2,c_2,h_2) \text{ :- } load(v_1,f,v_2), vP_{\mathrm{T}}(c,v_1,c_1,h_1), hP_{\mathrm{T}}(c_1,h_1,f,c_2,h_2), vPfilter(v_2,h_2). \quad (30)$$

□

H_T: C × H encodes the non-thread objects created by a thread. $H_T(c, h)$ means that a thread with context c may execute non-thread allocation site h; in other words, there is a call path from the `run()` method in context c to allocation site h.

vP_{0T}: C × V × C × H is the set of initial inter-thread points-to relations. This includes the points-to relations for thread creation sites and for the global object. $vP_{0T}(c_1, v, c_2, h)$ means that thread c_1 has an thread allocation site h, and v points to the newly created thread context c_2. (There are usually two contexts assigned to each allocation site). All global objects across all contexts are given the same context.

vP_T: C × V × C × H is the thread-sensitive version of the variable points-to relation vP_C. $vP_T(c_1, v, c_2, h)$ means variable v in context c_1 can point to heap object h created under context c_2.

hP_T: C × H × F × C × H is the thread-sensitive version of the heap points-to relation hP. $hP_T(c_1, h_1, f, c_2, h_2)$ means that field f of heap object h_1 created under context c_1 can point to heap object h_2 created under context c_2.

Rule (26) incorporates the initial points-to relations for thread creation sites. Rule (27) incorporates the points-to information for non-thread creation sites, which have the context numbers of threads that can reach the method. The other rules are analogous to those of the context-sensitive pointer analysis, with an additional context attribute for the heap objects.

From the analysis results, we can easily determine which objects have escaped. An object h created by thread context c has escaped, written $escaped(c, h)$, if it is accessed by a different context c_v. Complications involving unknown code, such as native methods, could also be handled using this technique.

$$escaped(c, h) \quad :- \quad vP_T(c_v, \_, c, h), c_v \neq c.$$

Conversely, an object h created by context c is captured, written $captured(c, h)$, if it has not escaped. Any captured object can be allocated on a thread-local heap.

$$captured(c, h) \quad :- \quad vP_T(c, v, c, h), \neg escaped(c, h).$$

We can also use escape analysis to eliminate unnecessary synchronizations. We define a relation $syncs(v)$ indicating if the program contains a synchronization operation performed on variable v. A synchronization for variable v under context c is necessary, written $neededSyncs(c, v)$, if $syncs(v)$ and v can point to an escaped object.

$$neededSyncs(c, v) \quad :- \quad syncs(v), vP_T(c, v, c_h, h), \\ escaped(c_h, h)$$

Notice that $neededSyncs$ is context-sensitive. Thus, we can distinguish when a synchronization is necessary only for certain threads, and generate specialized versions of methods for those threads.

6. EXPERIMENTAL RESULTS

In this section, we present some experimental results of using bddbddb on the Datalog algorithms presented in this paper. We describe our testing methodology and benchmarks, present the analysis times, evaluate the results of the analyses, and provide some insight on our experience of developing these analyses and the bddbddb tool.

6.1 Methodology

The input to bddbddb is more or less the Datalog programs exactly as they are presented in this paper. (We added a few rules to handle return values and threads, and added annotations for the physical domain assignments of input relations.) The input relations were generated with the Joeq compiler infrastructure[32]. The entire bddbddb implementation is only 2500 lines of code. bddbddb uses the JavaBDD library[31], an open-source library based on the BuDDy library[21]. The entire system is available as open-source[35], and we hope that others will find it useful.

All experiments were performed on a 2.2GHz Pentium 4 with Sun JDK 1.4.2_04 running on Fedora Linux Core 1. For the context-insensitive and context-sensitive experiments, respectively: we used initial BDD table sizes of 4M and 12M; the tables could grow by 1M and 3M after each garbage collection; the BDD operation cache sizes were 1M and 3M.

To test the scalability and applicability of the algorithm, we applied our technique to 21 of the most popular Java projects on Sourceforge as of November 2003. We simply walked down the list of 100% Java projects sorted by activity, selecting the ones that would compile directly as standalone applications. They are all real applications with tens of thousands of users each. As far as we know, these are the largest benchmarks ever reported for any context-sensitive Java pointer analysis. As a point of comparison, the largest benchmark in the specjvm suite, javac, would rank only 13th in our list.

For each application, we chose an applicable `main()` method as the entry point to the application. We included all class initializers, thread run methods, and finalizers. We ignored null constants in the analysis—every points-to set is automatically assumed to include null. Exception objects of the same type were merged. We treated reflection and native methods as returning unknown objects. Some native methods and special fields were modeled explicitly.

A short description of each of the benchmarks is included in Figure 3, along with their vital statistics. The number of classes, methods, and bytecodes were those discovered by the context-insensitive on-the-fly call graph construction algorithm, so they include only the reachable parts of the program and the class library.

The number of context-sensitive (C.S.) paths is for the most part correlated to the number of methods in the program, with the exception of pmd. pmd has an astounding 5×10^{23} paths in the call graph, which requires 79 bits to represent. pmd has different characteristics because it contains code generated by the parser generator JavaCC. Many machine-generated methods call the same class library routines, leading to a particularly egregious exponential blowup. The JavaBDD library only supports physical domains up to 63 bits; contexts numbered beyond 2^{63} were merged into a single context. The large number of paths also caused the algorithm to require many more rule applications to reach a fixpoint solution.

6.2 Analysis Times

We measured the analysis times and memory usage for each of the algorithms presented in this paper (Figure 4). The algorithm with call graph discovery, in each iteration, computes a call graph based on the points-to relations from the previous iteration. The number of iterations taken for that algorithm is also included here.

All timings reported are wall-clock times from a cold start, and include the various overheads for Java garbage collection, BDD garbage collection, growing the node table, etc. The memory numbers reported are the sizes of the peak number of live BDD nodes during the course of the algorithm. We measured peak BDD memory usage by setting the initial table size and maximum table size increase to 1MB, and only allowed the table to grow if the node table was more than 99% full after a garbage collection.[5]

[5]To avoid garbage collections, it is recommended to use more memory. Our timing runs use the default setting of 80%.

Name	Description	Classes	Methods	Bytecodes	Vars	Allocs	C.S. Paths
freetts	speech synthesis system	215	723	48K	8K	3K	4×10^4
nfcchat	scalable, distributed chat client	283	993	61K	11K	3K	8×10^6
jetty	HTTP Server and Servlet container	309	1160	66K	12K	3K	9×10^5
openwfe	java workflow engine	337	1215	74K	14K	4K	3×10^6
joone	Java neural net framework	375	1531	92K	17K	4K	1×10^7
jboss	J2EE application server	348	1554	104K	17K	4K	3×10^8
jbossdep	J2EE deployer	431	1924	119K	21K	5K	4×10^8
sshdaemon	SSH daemon	485	2053	115K	24K	5K	4×10^9
pmd	Java source code analyzer	394	1971	140K	19K	4K	5×10^{23}
azureus	Java bittorrent client	498	2714	167K	24K	5K	2×10^9
freenet	anonymous peer-to-peer file sharing system	667	3200	210K	38K	8K	2×10^7
sshterm	SSH terminal	808	4059	241K	42K	8K	5×10^{11}
jgraph	mathematical graph-theory objects and algorithms	1041	5753	337K	59K	10K	1×10^{11}
umldot	makes UML class diagrams from Java code	1189	6505	362K	65K	11K	3×10^{14}
jbidwatch	auction site bidding, sniping, and tracking tool	1474	8262	489K	90K	16K	7×10^{13}
columba	graphical email client with internationalization	2020	10574	572K	111K	19K	1×10^{13}
gantt	plan projects using Gantt charts	1834	10487	597K	117K	20K	1×10^{13}
jxplorer	ldap browser	1927	10702	645K	133K	22K	2×10^9
jedit	programmer's text editor	1788	10934	667K	124K	20K	6×10^7
megamek	networked BattleTech game	1265	8970	668K	123K	21K	4×10^{14}
gruntspud	graphical CVS client	2277	12846	687K	145K	24K	2×10^9

Figure 3: Information about the benchmarks we used to test our analyses.

The context-insensitive analyses (Algorithms 1 and 2) are remarkably fast; the type-filtering version was able to complete in under 45 seconds on all benchmarks. It is interesting to notice that introducing type filtering actually improved the analysis time and memory usage. Along with being more accurate, the points-to sets are much smaller in the type-filtered version, leading to faster analysis times.

For Algorithm 3, the call graph discovery sometimes took over 40 iterations to complete, but it was very effective in reducing the size of the call graph as compared to CHA[19]. The complexity of the call graph discovery algorithm seems to vary with the number of virtual call sites that need resolving—jedit and megamek have many methods declared as final, but jxplorer has none, leading to more call targets to resolve and longer analysis times.

The analysis times and memory usages of our context-sensitive points-to analysis (Algorithm 5) were, on the whole, very reasonable. It can analyze most of the small and medium size benchmarks in a few minutes, and it successfully finishes analyzing even the largest benchmarks in under 19 minutes. This is rather remarkable considering that the context-sensitive formulation is solving up to 10^{14} times as many relations as the context-insensitive version! Our scheme of numbering the contexts consecutively allows the BDD to efficiently represent the similarities between calling contexts. The analysis times are most directly correlated to the number of paths in the call graph. From the experimental data presented here, it appears that the analysis time of the context-sensitive algorithm scales approximately with $O(\lg^2 n)$ where n is the number of paths in the call graph; more experiments are necessary to determine if this trend persists across more programs.

The context-sensitive type analysis (Algorithm 6) is, as expected, quite a bit faster and less memory-intensive than the context-sensitive points-to analysis. Even though it uses the same number of contexts, it is an order of magnitude faster than the context-sensitive points-to analysis. This is because in the type analysis the number of objects that can be pointed to is much smaller, which greatly increases sharing in the BDD. The thread-sensitive pointer analysis (Algorithm 7) has analysis times and memory usages that are roughly comparable to those of the context-insensitive pointer analysis, even though it includes thread context information. This is because the number of thread creation sites is relatively small, and we use at most two contexts per thread.

6.3 Evaluation of Results

An in-depth analysis of the accuracy of the analyses with respect to each of the queries in Section 5 is beyond the scope of this paper. Instead, we show the results of two specific queries: thread escape analysis (Section 5.6) and type refinement (Section 5.3).

The results of the escape analysis are shown in Figure 5. The first two columns give the number of captured and escaped object creation sites, respectively. The next two columns give the number of unneeded and needed synchronization operations. The single-threaded benchmarks have only one escaped object: the global object from which static variables are accessed. In the multi-threaded benchmarks, the analysis is effective in finding 30-50% of the allocation sites to be captured, and 15-30% of the synchronization operations to be unnecessary. These are static numbers; to fully evaluate the results would require dynamic execution counts, which is outside of the scope of this paper.

The results of the type refinement query are shown in Figure 6. We tested the query across six different analysis variations. From left to right, they are context-insensitive pointer analysis without and with type filtering, context-sensitive pointer analysis and context-sensitive type analysis with the context projected away, and context-sensitive pointer and type analysis on the fully cloned graph. Projecting away the context in a context-sensitive analysis makes the result context-insensitive; however, it can still be more precise than context-insensitive analysis because of the extra precision at the intermediate steps of the analysis. We measured the percentages of variables that can point to multiple types and variables whose types can be refined.

Including the type filtering makes the algorithm strictly more precise. Likewise, the context-sensitive pointer analysis is strictly more precise than both the context-insensitive pointer analysis and the context-sensitive type analysis. We can see this trend in the results. As the precision increases, the percentage of multi-typed variables drops and the percentage of refinable variables increases. The context-insensitive pointer analysis and the context-sensitive type analysis are not directly comparable; in some cases the point-

Name	Context-insensitive pointers							Context-sensitive				Thread-sensitive pointer analysis	
	no type filter		with type filter		with cg discovery			pointer analysis		type analysis			
	time	mem	time	mem	iter	time	mem	time	mem	time	mem	time	mem
freetts	1	3	1	3	20	2	4	1	6	1	6	1	4
nfcchat	1	4	1	4	23	4	6	2	12	2	12	1	6
jetty	1	5	1	5	22	4	7	3	12	2	10	1	6
openwfe	1	5	1	6	23	5	8	4	14	2	14	1	7
joone	2	7	1	7	24	7	10	4	18	3	18	1	9
jboss	2	7	2	7	30	8	10	7	24	4	22	2	9
jbossdep	2	9	2	9	26	7	12	9	30	5	26	3	11
sshdaemon	2	9	2	10	26	13	14	12	34	6	28	3	13
pmd	1	7	1	7	33	9	10	297	111	19	36	1	9
azureus	2	10	2	10	29	13	15	9	32	6	30	2	12
freenet	7	16	5	16	40	41	23	21	38	10	32	6	21
sshterm	8	17	5	17	31	37	25	50	86	18	60	7	23
jgraph	17	27	11	25	42	78	37	119	134	33	20	13	35
umldot	17	30	11	29	34	97	43	457	304	63	130	16	41
jbidwatch	31	43	20	40	32	149	58	580	394	68	140	25	56
columba	43	55	27	49	42	273	73	807	400	123	178	38	72
gantt	41	59	26	51	39	261	76	1122	632	113	174	34	71
jxplorer	57	68	39	60	41	390	88	337	198	78	118	51	83
jedit	61	61	38	54	37	278	80	113	108	60	82	50	76
megamek	40	57	26	51	34	201	76	1101	600	100	224	34	73
gruntspud	66	76	41	67	35	389	99	312	202	86	130	58	95

Figure 4: Analysis times and peak memory usages for each of the benchmarks and analyses. Time is in seconds and memory is in megabytes.

ers are more precise, in other cases the context-sensitive types are more precise.

When we do not project away the context, the context-sensitive results are remarkably precise—the percentage of multi-typed variables is never greater than 1% for the pointer analysis and 2% for the type analysis. Projecting away the context loses much of the benefit of context sensitivity, but is still noticeably more precise than using a context-insensitive analysis.

Name	heap objects		sync operations	
	captured	*escaped*	$\neg needed$	*needed*
freetts	2349	1	43	0
nfcchat	1845	2369	52	46
jetty	2059	2408	47	89
openwfe	3275	1	57	0
joone	1640	1908	34	75
jboss	3455	2836	112	105
jbossdep	1838	2298	32	94
sshdaemon	12822	22669	468	1244
pmd	3428	1	47	0
azureus	8131	9183	226	229
freenet	5078	9737	167	309
sshterm	16118	24483	767	3642
jgraph	25588	48356	1078	5124
umldot	38930	69332	2146	8785
jbidwatch	97234	143384	2243	11438
columba	111578	174329	3334	18223
gantt	106814	156752	2377	11037
jxplorer	188192	376927	4127	18904
jedit	446896	593847	7132	36832
megamek	179221	353096	3846	22326
gruntspud	248426	497971	5902	25568

Figure 5: Results of escape analysis.

6.4 Experience

All the experimental results reported here are generated using bddbddb. At the early stages of our research, we hand-coded every points-to analysis using BDD operations directly and spent a considerable amount of time tuning their performance. Our context-numbering scheme is the reason why the analysis would finish at all on even small programs. Every one of the optimizations described in Section 2.4 was first carried out manually. After considerable effort, megamek still took over three hours to analyze, and jxplorer did not complete at all. The incrementalization was very difficult to get correct, and we found a subtle bug months after the implementation was completed. We did not incrementalize the outermost loops as it would have been too tedious and error-prone. It was also difficult to experiment with different rule application orders.

To get even better performance, and more importantly to make it easier to develop new queries and analyses, we created bddbddb. We automated and extended the optimizations we have used in our manual implementation, and implemented a few new ones to empirically choose the best BDD library parameters. The end result is that code generated by bddbddb outperforms our manually tuned context-sensitive pointer analysis by as much as an order of magnitude. Even better, we could use bddbddb to quickly and painlessly develop new analyses that are highly efficient, such as the type analysis in Section 5.5 and the thread escape analysis in Section 5.6.

7. RELATED WORK

This paper describes a scalable cloning-based points-to analysis that is context-sensitive, field-sensitive, inclusion-based and implemented using BDDs. Our program analyses, expressed in Datalog, are translated by bddbddb into a BDD implementation automatically. We also presented example queries using our system to check for vulnerabilities, infer types, and find objects that escape a thread. Due to space constraints, we can only describe work that is very closely related to ours.

Scalable pointer analyses. Most of the scalable algorithms proposed are context-insensitive and flow-insensitive. The first scalable pointer analysis proposed was a unification-based algorithm due to Steensgaard[28]. Das et al. extended the unification-based approach to include "one-level-flow"[10] and one level of context sensitivity[11]. Subsequently, a number of inclusion-based algorithms have been shown to scale to large programs[3, 17, 19, 33].

A number of context-sensitive but flow-insensitive analyses have been developed recently[15, 16]. The C pointer analysis due to

Name	Context-insensitive pointers				Projected context-sensitive				Context-sensitive			
	no type filter		with type filter		pointer analysis		type analysis		pointer analysis		type analysis	
	multi	refine	multi	refine	multi	refine	multi	refine	multi	refine	multi	refine
freetts	5.1	41.1	2.3	41.6	2.0	41.9	2.5	41.3	0.1	44.4	0.3	44.0
nfcchat	12.4	36.4	8.6	37.0	8.2	37.4	8.6	36.9	0.1	45.9	0.7	45.3
jetty	12.6	36.2	7.7	37.1	7.3	37.4	7.7	37.1	0.1	45.4	0.6	44.8
openwfe	12.1	36.9	7.0	37.7	6.6	38.0	7.0	37.6	0.1	45.5	0.5	44.8
joone	11.9	37.5	6.8	38.1	6.4	38.4	6.7	38.1	0.1	45.8	0.5	45.0
jboss	13.4	37.8	7.9	38.7	7.4	39.3	7.8	38.7	0.1	47.3	0.7	46.4
jbossdep	10.2	40.3	7.4	39.5	7.0	40.0	7.5	39.4	0.2	47.6	0.8	46.6
sshdaemon	10.7	39.3	6.0	40.3	5.8	40.5	5.9	40.4	0.1	46.8	0.6	46.1
pmd	9.6	42.3	6.2	43.1	5.9	43.4	6.2	43.1	0.1	52.1	0.6	48.1
azureus	10.0	43.6	6.1	44.1	6.0	44.3	6.2	44.1	0.1	50.8	0.9	49.7
freenet	12.1	39.1	6.3	40.0	5.9	40.5	6.3	40.1	0.1	47.0	0.8	46.0
sshterm	14.7	40.8	8.9	42.0	8.5	42.5	9.0	42.1	0.6	51.3	1.6	49.9
jgraph	16.1	43.2	9.6	45.1	9.3	45.4	9.7	45.2	0.7	54.7	1.9	53.2
umldot	15.7	42.3	9.4	43.6	9.0	43.9	9.4	43.6	0.6	53.0	2.0	51.2
jbidwatch	14.9	42.3	8.6	43.4	8.2	43.7	8.6	43.4	0.6	52.0	1.7	50.5
columba	15.7	42.3	9.0	43.7	8.6	44.1	8.9	43.9	0.6	52.4	1.8	51.0
gantt	15.0	43.4	8.2	44.7	7.9	45.0	8.2	44.7	0.5	53.0	1.7	51.4
jxplorer	15.2	43.1	7.9	44.3	7.7	44.6	8.0	44.4	0.5	52.5	1.6	50.8
jedit	15.4	43.6	8.1	44.7	7.9	44.9	8.1	44.7	0.6	53.1	1.6	51.5
megamek	13.3	44.6	7.1	45.1	6.8	45.3	7.2	45.2	0.5	53.3	1.4	51.6
gruntspud	15.4	44.0	7.7	45.5	7.5	45.7	7.8	45.5	0.5	53.6	1.4	52.1

Figure 6: Results of the type refinement query. Numbers are percentages. Columns labeled multi and refine refer to multi-type variables and refinable-type variables, respectively.

Fähndrich et al.[15] has been demonstrated to work on a 200K-line `gcc` program. Unlike ours, their algorithm is unification-based and field-independent, meaning that fields in a structure are modeled as having the same location. Their context-sensitive analysis discovers targets of function pointers on-the-fly. Our algorithm first computes the call graph using a context-insensitive pointer alias analysis; there are significantly more indirect calls in Java programs, the target of our technique, due to virtual method invocations. Their algorithm uses CFL-reachability queries to implement context sensitivity[24]. Instead of computing context-sensitive solutions on demand, we compute all the context-sensitive results and represent them in a form convenient for further analysis.

Other context-sensitive pointer analysis. Some of the earlier attempts of context-sensitive analysis are flow-sensitive[14, 18, 34, 37]. Our analysis is similar to the work by Emami et al. in that they also compute context-sensitive points-to results directly for all the different contexts. Their analysis is flow-sensitive; ours uses flow sensitivity only in summarizing each method intraprocedurally. While our technique treats all members of a strongly connected component in a call graph as one unit, their technique only ignores subsequent invocations in recursive cycles. On the other hand, their technique has only been demonstrated to work for programs under 3000 lines.

As discussed in Section 1, using summaries is another common approach to context sensitivity. It is difficult to compute a compact summary if a fully flow-sensitive result is desired. One solution is to use the concept of partial transfer functions, which create summaries for observed calling contexts[36, 37]. The same summary can be reused by multiple contexts that share the same relevant alias patterns. This technique has been shown to handle C programs up to 20,000 lines.

One solution is to allow only weak updates[34]; that is, a write to a variable only adds a value to the contents of the variable without removing the previously held value. This greatly reduces the power of a flow-sensitive analysis. This approach has been used to handle programs up to 70,000 lines of code. However, on larger programs the representation still becomes too large to deal with. Because the goal of the prior work was escape analysis, it was not necessary to maintain precise points-to relations for locations that escape, so the algorithm achieved scalability by collapsing escaped nodes.

BDD-based pointer analysis. BDDs have recently been used in a number of program analyses such as predicate abstraction[2], shape analysis[23, 38], and, in particular, points-to analysis[3, 39]. Zhu proposed a summary-based context-sensitive points-to analysis for C programs, and reported preliminary experimental results on C programs with less than 5000 lines[39]. Berndl et al. showed that BDDs can be used to compute context-insensitive inclusion-based points-to results for large Java programs efficiently. In the same conference this paper is presented, Zhu and Calman describe a cloning-based context-sensitive analysis for C pointers, assuming that only the safe C subset is used. The largest program reported in their experiment has about 25,000 lines and 3×10^8 contexts[40].

High-level languages and tools for program analysis. The use of Datalog and other logic programming languages has previously been proposed for describing program analyses[12, 25, 30]. Our `bddbddb` system implements Datalog using BDDs[35] and has been used to compute context-sensitive points-to results and other advanced analyses. Other examples of systems that translate program analyses and queries written in logic programming languages into implementations using BDDs include Toupie[9] and CrocoPat[4]. Jedd is a Java language extension that provides a relational algebra abstraction over BDDs[20].

8. CONCLUSION

This paper shows that, by using BDDs, it is possible to obtain efficient implementations of context-sensitive analyses using an extremely simple technique: We clone all the methods in a call graph, one per context of interest, and simply apply a context-insensitive analysis over the cloned graph to get context-sensitive results. By numbering similar contexts contiguously, the BDD is able to handle the exponential blowup of contexts by exploiting their commonalities. We showed that this approach can be applied to type inference, thread escape analysis and even fully context-sensitive points-to analysis on large programs.

This paper shows that we can create efficient BDD-based analyses easily. By keeping data and analysis results as relations, we

can express queries and analyses in terms of Datalog. The bddbddb system we have developed automatically converts Datalog programs into BDD implementations that are even more efficient than those we have painstakingly hand-tuned.

Context-sensitive pointer analysis is the cornerstone of deep program analysis for modern programming languages. By combining (1) context-sensitive points-to results, (2) a simple approach to context sensitivity, and (3) a simple logic-programming based query framework, we believe we have made it much easier to create advanced program analyses.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0086160 and an NSF Graduate Student Fellowship. We thank our anonymous referees for their helpful comments.

9. REFERENCES

[1] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query optimization. *Journal of Logic Programming*, 4(3):259–262, Sept. 1987.

[2] T. Ball and S. K. Rajamani. A symbolic model checker for boolean programs. In *Proceedings of the SPIN 2000 Workshop on Model Checking of Software*, pages 113–130, Aug. 2000.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, June 2003.

[4] 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.

[5] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, Sept. 1996.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[7] A. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[8] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, Nov. 1999.

[9] 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 *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, pages 75–91, Aug. 1993.

[10] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, June 2000.

[11] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Static Analysis Symposium*, pages 260–278, July 2001.

[12] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–126, May 1996.

[13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aug. 1995.

[14] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[15] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, June 2000.

[16] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the 7th International Static Analysis Symposium*, pages 175–198, Apr. 2000.

[17] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.

[18] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[19] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 153–169, April 2003.

[20] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[21] J. Lind-Nielsen. BuDDy, a binary decision diagram package. http://www.itu.dk/research/buddy/.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[23] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In *Proceedings of the 9th International Static Analysis Symposium*, pages 196–212, Sept. 2002.

[24] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages*, pages 49–61, Jan. 1995.

[25] T. W. Reps. *Demand Interprocedural Program Analysis Using Logic Databases*, pages 163–196. Kluwer, 1994.

[26] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[27] F. Somenzi. CUDD: CU decision diagram package release, 1998.

[28] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

[29] Java cryptography extension (JCE). http://java.sun.com/products/jce, 2003.

[30] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., volume II edition, 1989.

[31] J. Whaley. JavaBDD library. http://javabdd.sourceforge.net.

[32] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *Proceedings of the SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, June 2003.

[33] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, Sept. 2002.

[34] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference of Object Oriented Programming: Systems, Languages, and Applications*, pages 187–206, Nov. 1999.

[35] J. Whaley, C. Unkel, and M. S. Lam. A BDD-based deductive database for program analysis. http://suif.stanford.edu/bddbddb, 2004.

[36] R. P. Wilson. *Efficient, context-sensitive pointer analysis for C programs*. PhD thesis, Stanford University, Dec. 1997.

[37] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[38] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proceedings of the 9th International Static Analysis Symposium*, pages 69–84, Sept. 2002.

[39] J. Zhu. Symbolic pointer analysis. In *Proceedings of the International Conference in Computer-Aided Design*, pages 150–157, Nov. 2002.

[40] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.