Strictly Declarative Specification of Sophisticated Points-to Analyses

Martin Bravenboer Yannis Smaragdakis

Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003, USA
martin.bravenboer@acm.org yannis@cs.umass.edu

Abstract

We present the DOOP framework for points-to analysis of Java programs. DOOP builds on the idea of specifying pointer analysis algorithms declaratively, using Datalog: a logic-based language for defining (recursive) relations. We carry the declarative approach further than past work by describing the full end-to-end analysis in Datalog and optimizing aggressively using a novel technique specifically targeting highly recursive Datalog programs.

As a result, DOOP achieves several benefits, including full order-of-magnitude improvements in runtime. We compare DOOP with Lhoták and Hendren's PADDLE, which defines the state of the art for context-sensitive analyses. For the exact same logical points-to definitions (and, consequently, identical precision) DOOP is more than 15x faster than PADDLE for a 1-call-site sensitive analysis of the DaCapo benchmarks, with lower but still substantial speedups for other important analyses. Additionally, DOOP scales to very precise analyses that are impossible with PADDLE and Whaley et al.'s bddbddb, directly addressing open problems in past literature. Finally, our implementation is modular and can be easily configured to analyses with a wide range of characteristics, largely due to its declarativeness.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; D.1.6 [*Programming Techniques*]: Logic Programming

General Terms Algorithms, Languages, Performance

1. Introduction

Points-to (or *pointer*) analysis intends to answer the question "what objects can a program variable point to?" This question forms the basis for practically all higher-level program analyses. It is, thus, not surprising that a wealth of research has been devoted to efficient and precise pointer analysis techniques. *Context-sensitive* analyses are the most common class of precise points-to analyses. Context sensitive analysis approaches qualify the analysis facts with a *context* abstraction, which captures a static notion of the dynamic context of a method. Typical contexts include abstractions of method call-sites (for a *call-site sensitive* analysis—the traditional meaning of "context-sensitive") or receiver objects (for an *object-sensitive* analysis).

In this work we present DOOP: a general and versatile points-to analysis framework that makes feasible the most precise context-sensitive analyses reported in the literature. DOOP implements a range of algorithms, including context insensitive, call-site sensitive, and object-sensitive analyses, all specified modularly as variations on a common code base. Compared to the prior state of the art, DOOP often achieves speedups of an order-of-magnitude for several important analyses.

The main elements of our approach are the use of the Datalog language for specifying the program analyses, and the aggressive optimization of the Datalog program. The use of Datalog for program analysis (both low-level [13,23,29] and high-level [6,9]) is far from new. Our novel optimization approach, however, accounts for several orders of magnitude of performance improvement: unoptimized analyses typically run over 1000 times more slowly. Generally our optimizations fit well the approach of handling program facts as a database, by specifically targeting the indexing scheme and the incremental evaluation of Datalog implementations. Furthermore, our approach is entirely Datalog based, encoding declaratively the logic required both for call graph construction as well as for handling the full semantic complexity of the Java language (e.g., static initialization, finalization, reference objects, threads, exceptions, reflection, etc.). This makes our pointer analysis specifications elegant, modular, but also efficient and easy to tune. Generally, our work is a strong data point in support of declarative languages: we argue that prohibitively much human effort is required for implementing and optimizing complex mutually-recursive definitions at an operational level of abstraction. On the other

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

Copyright © 2009 ACM 978-1-60558-734-9/09/10. . . $5.00

hand, declarative specifications both admit automatic optimizations as well as afford the user the ability to identify and apply straightforward manual optimizations.

We evaluate Doop in comparison to Lhoták and Hendren's Paddle framework [18]. Paddle is based on Binary Decision Diagrams (BDDs) and represents the state of the art in context sensitive pointer analyses, in terms of both semantic completeness (i.e., support for Java language features) and scalability. Furthermore, Paddle is a highly flexible framework that was used to illustrate the different characteristics and parameters of context-sensitivity. Doop has the same attractive features and yields identical analysis results (based on a logically equivalent algorithm). Our 1-call-site-sensitive analysis of the DaCapo benchmarks applications (and JDK 1.4) yields an average speedup of 16.3x, lowering analysis times from several minutes to below a minute in many cases. For a 1-object-sensitive analysis, Doop is 15x faster than Paddle. Such speedups are rare in the program analysis literature, especially for completely equivalent analyses. Lhoták and Hendren recently speculated [18] *"it should be feasible to implement an efficient non-BDD-based 1-object-sensitive analysis"*. We show that such an analysis not only is feasible but also outperforms BDDs by an order of magnitude.

Generally, our approach reveals interesting insights regarding the use of BDDs, compared to an *explicit representation* of relations, for points-to analysis. Our work raises the question of whether the points-to analysis domain has enough regularity for BDDs to be beneficial. Although we have found analyses that are possible with BDDs yet we could not perform with an explicit representation, every such analysis seemed to suffer from vast (but very regular) imprecision. Easy algorithmic enhancements can be applied to reduce the unnecessary redundancy in the relations that a points-to analysis keeps, and produce an analysis that is both much faster without BDDs and more precise. For instance, Doop would not scale to a 1H-object-sensitive analysis (i.e., 1-object-sensitive with a context-sensitive heap) in the form specified in the Paddle analysis set. Yet this is only because a naive analysis specification results in high redundancy, which necessitates BDDs. Two simple algorithmic enhancements suffice for making the analysis feasible for Doop: *1)* we perform exception analysis on-the-fly [3], computing contexts that are reachable because of exceptional control flow while performing the points-to analysis itself. The on-the-fly exception analysis significantly improves both precision and performance; *2)* we treat static class initializers context-insensitively (since points-to results are equivalent for all contexts of static class initializers), thus improving performance while keeping identical precision.

The result of combining Doop's optimization approach and our algorithmic enhancements is that Doop addresses several open problems in the points-to analysis literature. Lhoták and Hendren estimated that *"efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in the data structures and algorithms used to implement points-to analyses"* [18]. Doop achieves this goal, with fairly routine data structures (plain B-trees). Furthermore, Doop reproduces the most complex points-to analyses of the Paddle set—a result previously considered impossible without BDDs. Even more importantly, Doop scales to analyses that are impossible with current BDD-based approaches, such as a 2H-call-site-sensitive analysis.

In summary, our work makes the following contributions:

- We provide the first *fully declarative* specification of complex, highly precise points-to analyses. Our specification *distills points-to analysis algorithms down to their essence*, instead of confusing the logical statement of an analysis with implementation details. Past work on specifying points-to analyses in Datalog has always been a hybrid between imperative code and a logical specification, omitting essential elements from the logic. For instance, the bddbddb system [28, 29] (which pioneered practical Datalog-based points-to analysis) expresses the core of a points-to analysis in Datalog, while important parts (such as normalization and call-graph computation—except for simple, context-insensitive, analyses) are done in Java code. In general, Doop offers the first declarative specification of a context-sensitive points-to analysis with on-the-fly (i.e., fully interleaved) call-graph computation. Additionally, our specification of algorithms is quite sophisticated, addressing elements of the Java language (such as native code, finalization, and privileged actions) that were absent from previous declarative approaches (e.g., bddbddb) and that crucially affect precision and performance. As a result, Doop provides an analysis that emulates and often exceeds the rich feature set of the Paddle framework, while staying entirely declarative.
- We introduce a novel optimization methodology, applied entirely at the Datalog level, for producing efficient algorithms directly from the logical specification of an analysis. The optimization approach employs standard program transformations (such as variable reordering and folding—a common logic programming optimization) yet determines when to do so by taking into account the "semi-naive" algorithm for incremental evaluation of Datalog rules, as well as the indexes that are used for each relation. As a result, Doop achieves order-of-magnitude performance improvements over the closest comparable points-to framework in the literature for common context-sensitive analyses.
- We show that Doop scales to perform the most precise context-sensitive analyses ever evaluated in the research literature. Doop not only implements the rich set of analyses of the Paddle system but also scales to analyses that are beyond reach for Paddle, such as a 2-call-site-sensitive analysis with a context-sensitive heap, and

a 2-object-sensitive analysis with a (1-context) context-sensitive heap.

- We contrast and study the performance of BDD-based representations for points-to analysis, relative to explicit representations. We show how performance is correlated with key BDD metrics and extrapolate on the suitability of BDDs for fast and precise points-to analyses.

2. Background: Datalog Points-To Analysis

The use of deductive databases and logic programming languages for program analysis has a long history (e.g., [4, 23]) and has raised excitement again recently [6, 9, 13, 28, 29]. Like our work, much of the past emphasis has been on using the Datalog language. Datalog is a logic programming language originally introduced in the database domain. At a first approximation, one can view Datalog as either "SQL with full recursion" or "Prolog without constructors/functions". The essence of the language is its ability to define recursive relations. Relations (or equivalently *predicates*) are the main Datalog data type. Computation consists of inferring the contents of all relations from a set of input relations. For instance, in our pointer analysis domain, it is easy to represent the relevant actions of a Java program as relations, typically stored as database tables. Consider two such relations, `AssignHeapAllocation(?var,?heap)` and `Assign(?from,?to)`. (We follow the convention of capitalizing the first letter of relation names, while writing variable names in lower case and prefixing them with a question-mark.) The former relation represents all occurrences in the program of an instruction "`a = new A();`" where a heap object is allocated and assigned to a variable. That is, a pre-processing step takes a Java program (in our implementation this is in intermediate, bytecode, form) as input and produces the relation contents. A static abstraction of the heap object is captured in variable `?heap`—it can be concretely represented as, e.g., a fully qualified class name and the allocation's bytecode instruction index. Similarly, relation `Assign` contains an entry for each assignment between two Java program (reference) variables.

The mapping between the input Java program and the input relations is straightforward and purely syntactic. After this step, a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
1 VarPointsTo(?var, ?heap) <-
2    AssignHeapAllocation(?var, ?heap).
3 VarPointsTo(?to, ?heap) <-
4    Assign(?from, ?to), VarPointsTo(?from, ?heap).
```

The Datalog program consists of a series of *rules* that are used to establish facts about derived relations (such as `VarPointsTo`, which is the points-to relation, i.e., it links every program variable, `?var`, with every heap object abstraction, `?heap`, it can point to) from a conjunction of previously established facts. We use the left arrow symbol (`<-`) to separate the inferred fact (the *head*) from the previously established facts (the *body*). For instance, lines 3-4 above say that if, for some values of `?from`, `?to`, and `?heap`, `Assign(?from,?to)` and `VarPointsTo(?from,?heap)` are both true, then it can be inferred that `VarPointsTo(?to,?heap)` is true. Note the base case of the computation above (lines 1-2), as well as the recursion in the definition of `VarPointsTo` (line 3-4).

The declarativeness of Datalog makes it attractive for specifying complex program analysis algorithms. Particularly important is the ability to specify recursive definitions, as program analysis is fundamentally an amalgam of mutually recursive tasks. For instance, in order to do accurate reachability analysis (i.e., answer the question "is method `m1` reachable from method `m2`?") we need to have points-to information, so that the target objects of a virtual method call are known. But in order to do points-to analysis, we need to have reachability information, to know which variable assignment actions are truly possible. A mutually recursive definition of a reachability and points-to analysis is easy to specify in Datalog, and is part of the Doop framework. The elegance of the approach is evident when contrasted with common implementations of points-to analyses. Even conceptually clean program analysis algorithms that rely on mutually recursive definitions often get transformed into complex imperative code for implementation purposes (e.g., compare the straightforward logic with the complex algorithmic specification in Reference [26]).

Datalog evaluation is typically *bottom-up*, meaning that known facts are propagated using the rules until a maximal set of derived facts is reached. This is also the link to the data processing intended domain of Datalog: evaluation of a rule can be thought of as a sequence of relational algebra joins and projections. For instance, the evaluation of lines 3-4 in our above example can be thought of as: Take the join of relation `Assign` with relation `VarPointsTo` over the first column of both (because of common field `?from`) and project the join result on fields `?to` and `?heap`. The result of the projection is added to relation `VarPointsTo` (skipping duplicates) and forms the value of `VarPointsTo` for the next iteration step. Application of all rules iterates to fixpoint. Note that this means that the evaluation of a Datalog program comprises two distinct kinds of looping/iteration activities: the relational algebra joins and projections, and the explicit recursion of the program. The former kind of looping is highly efficient through traditional database optimizations (e.g., for join order, group-fetching of data from disk, locality of reference, etc.).

We use a commercial Datalog engine, developed by our industrial partner, LogicBlox Inc. (The engine is freely available for research use through us and we have already granted access to a handful of early adopters.) This version of Datalog allows "stratified negation", i.e., negated clauses, as long as the negation is not part of a recursive cycle. It also allows

specifying that some relations are functions, i.e., the variable space is partitioned into domain and range variables, and there is only one range value for each unique combination of values in domain variables. We will see these features in action in our algorithm specification, next.

3. Doop Pointer Analysis Specifications

Our Doop framework is a versatile Datalog implementation of a range of pointer analyses. Doop is available online at `http://doop.program-analysis.org`. Doop strives for full Java language support and follows closely the approach of Paddle—the most complete analysis in prior literature—in dealing with various Java language features. We next discuss in more detail the features and precision of the framework.

3.1 Overview and Preliminaries

Doop distills points-to analysis algorithms to a purely declarative specification. An advantage of a declarative specification is that it dissociates the logic of the analysis (i.e., the precision of the end result as well as intermediate results) from the implementation decisions used to perform the analysis efficiently. The resulting specification is a Datalog program, and is, therefore, executable. Nevertheless, the program may not be efficient as originally specified. The goal of our optimization methodology (described in Section 4) is to produce equivalent Datalog programs that are more efficient. For the rest of this section, however, we are only concerned with the logical specification of the analyses.

This separation of specification from implementation is already done informally, as a classification, in the points-to analysis literature. Several different published algorithms occupy the same point in the design space (e.g., they are all 1-object-sensitive analyses) but differ in properties such as their average runtime or asymptotic complexity, often because of different choices of indexing and storage data structures. Hence, the effort to specify an analysis in Doop consists of, first, producing a logical specification and, then, deriving an efficient algorithm for that specification. The two steps are not entirely independent, because it is sometimes hard to tell which decisions are part of the "specification" of an analysis and which are part of the "implementation". For example, treating the static initializers of Java classes context-insensitively (even for a context-sensitive analysis) does not affect the end result of an analysis, but has a major impact on its runtime. However, expressing this decision affects the specification: the two Datalog programs are not equivalent. (What makes these specifications equivalent is extra knowledge about the input relations, i.e., a restriction of the input domain that only the algorithm designer knows.) In this paper, we call *optimizations* the transformations that produce an equivalent Datalog program (i.e., all relations have the same contents for all inputs), and call *logical enhancements* or *algorithmic enhancements* the transformations that logically change the original specification.

3.2 Doop Contents

Doop supports a general pointer analysis trunk and several different analysis variations. The main variants we have explored are a context-insensitive analysis, as well as context sensitive analyses with 1- and 2-object, as well as 1- and 2-call-site contexts, with or without a context-sensitive heap (a.k.a. *heap cloning*) with different heap context depths. This variability is not directly supported in Datalog: for instance, for a context-sensitive analysis, the relation `VarPointsTo` needs extra arguments representing the context (be it a call-site context, or an object context) of the variable. Similarly, for analyses utilizing a context-sensitive heap, the abstraction of the heap object needs to be qualified by extra variables for its context. (In such analyses, an abstract object consists of the allocation site and the context of the method that contains that allocation site.) These differences are superficial, however. We have abstracted away from them by creating a small extension of Datalog that allows tuples of variables in place of a single variable. The extension is implemented as a macro and hides the configuration of the particular analysis, to the extent possible. The plain-Datalog code for each analysis is then generated by instantiating the macros. The total size of the analysis logic in Doop is less than 2500 lines of code (approximately 180 Datalog program rules) and another some 1000 lines of relation declarations (i.e., specifications of the database schema), comments, and minor support code. These metrics include all pointer analysis variants, but commonalities are factored out using our variable-tuple mechanism. The plain-Datalog size of a single analysis variant after macro-expansion is in the order of 500-1000 lines, or 120-150 Datalog rules. In the code examples of this paper, unless stated otherwise, we will ignore variations and concentrate on the standard 1-call-site-sensitive analysis for concreteness.

The analysis logic in Doop can be viewed as an elaboration of the simple Datalog example shown earlier. Consider the full-fledged analogues of the two basic rules from Section 2.

```
1 VarPointsTo(?ctx, ?var, ?heap) <-
2    AssignHeapAllocation(?var, ?heap, ?inmethod),
3    CallGraphEdge(_, _, ?ctx, ?inmethod).
4
5 VarPointsTo(?toCtx, ?to, ?heap) <-
6    Assign(?fromCtx, ?from, ?toCtx, ?to, ?type),
7    VarPointsTo(?fromCtx, ?from, ?heap),
8    HeapAllocation:Type[?heap] = ?heaptype,
9    AssignCompatible(?type, ?heaptype).
```

(We use some extensions and notational conventions in the code. First, some of our relations are functions, and the functional notation "`Relation[?domainvar] = ?val`" is used instead of the relational notation, "`Relation(?domainvar, ?val)`". Semantically the two are equivalent, but the execution engine enforces the functional constraint and produces an error if a computation causes a function to have multiple

range values for the same domain value. Second, the colon (:) in relation names is just a regular character with no semantic significance—we use common prefixes ending with a colon as a lexical convention for grouping related predicates. Finally, "_" stands for "any value", in the standard logic programming convention.)

The full rules differ from their simplified versions in several ways. First, all relations have extra arguments for the context of Java variables: wherever the original relations had a Datalog variable that corresponded to a Java program variable (e.g., `?from`, `?to`) the full relations have first a Datalog variable corresponding to a context, and then one corresponding to the Java variable. Second, for an allocation to flow to a variable in a given context, the allocation site has to be reachable in the given context, from any other method and context (line 3). Finally, variable assignments take into account the type system (through `AssignCompatible`, on line 9) so that a variable is never considered to point to an object abstraction if its type prohibits it.

Some more rules complete the definition of `VarPointsTo`. The full analysis takes into account method calling, assignment to fields, arrays, and more.

Importantly, the entire analysis is specified in Datalog, including call graph construction. That is, the interdependency between call graph construction (i.e., which methods are reachable in a given context) and points-to analysis is expressed as plain Datalog mutual recursion. This allows call graph discovery on-the-fly, which Lhoták and Hendren [18] find to be an important asset for precision. Previous pointer analysis algorithms in Datalog (mainly Whaley et al.'s bddbddb and its client analyses [21, 28, 29]) did not support on-the-fly call-graph discovery, except for very simple, context-insensitive, analyses.[1]

For a concrete instance of the mutual recursion, we can look at one of the rules defining the `CallGraphEdge` relation (which is used to compute `VarPointsTo` and itself uses `VarPointsTo`). The rule computes call graph edges due to virtual method invocations and is shown in Figure 1. The definition of `CallGraphEdge` also uses an auxiliary definition, shown in Figure 2, of a virtual method lookup relation.

Combined, this is the declarative specification of fully on-the-fly call graph discovery, which is more precise than a pre-computed call graph, as in bddbddb.

3.3 Support for Java Language Features

Doop offers full support for Java language semantics, entirely in Datalog, without other peripheral analyses. We closely modeled the handling of Java features after the logic in the Paddle system. Paddle covers several complex Java features and semantic complexities (e.g., finalization, privileged actions, threads, etc.). Implementing an analysis that is logically equivalent to Paddle helps demonstrate that our Datalog-based approach is a full-featured implementation and not a toy or a proof-of-concept.

Indeed, in several cases we found ways to add more precision or model Java semantics better than Paddle, thus improving over past state-of-the-art techniques and making Doop probably the most sophisticated pointer analysis framework in existence for Java. This sophistication is important for client analyses that need sound results. For instance, compared to Paddle, Doop adds such features as:

- Better initialization of the Java Virtual Machine. For example, we model the system and main thread group, main thread.
- Full support for Java's reference objects (such as `WeakReference`) and reference queues. For example, ref-

[1] Specifically, the bddbddb work [13, 29] computes the call-graph on-the-fly with a context-insensitive analysis, and then uses it as input to context-sensitive analyses. Thus, the added precision of the context-sensitive points-to analysis is not available to the call-graph computation, which, in turn, reduces the precision of the points-to analysis. This limitation is not incidental. Since context-sensitivity in bddbddb is handled through a cloning approach [29], a pre-computed call-graph is necessary: cloning techniques are based on copying methods for each of their calling contexts.

```
CallGraphEdge(?callerCtx, ?call, ?calleeCtx, ?callee) <-
  VirtualMethodCall:Base[?call] = ?base,
  VirtualMethodCall:SimpleName[?call] = ?name,
  VirtualMethodCall:Descriptor[?call] = ?descriptor,
  VarPointsTo(?callerCtx, ?base, ?heap),
  HeapAllocation:Type[?heap] = ?heaptype,
  MethodLookup[?name, ?descriptor, ?heaptype] = ?callee,
  ?calleeCtx = ?call.
```

Figure 1. Computing (context-sensitive) call graph edges from a call-site to a method, both under specific contexts. A call graph edge exists if there exists a virtual method call, `?call`, whose receiver object is referenced through variable `?base`, which points to a heap object, `?heap`, whose type contains a method, `?callee`, compatible with the virtual call. The context of `?callee` for this call is just the call-site, `?call`, since the code is for a 1-call-site-sensitive analysis.

```
MethodLookup[?name, ?descriptor, ?type] = ?method <-
  MethodImpl[?name, ?descriptor, ?type] = ?method.

MethodLookup[?name, ?descriptor, ?type] = ?method <-
  DirectSuperclass[?type] = ?supertype,
  MethodLookup[?name, ?descriptor, ?supertype] = ?method,
  not exists MethodImpl[?name, ?descriptor, ?type].

MethodImpl[?name, ?descriptor, ?type] = ?method <-
  MethodDecl[?name, ?descriptor, ?type] = ?method,
  not MethodModifier("abstract", ?method).
```

Figure 2. The definition of relation `MethodLookup`, used in Figure 1. Looking up a method with a specific name (`?name`), return type, and parameter types (`?descriptor`) in a given type (`?type`) is done by either finding a non-abstract method declaration within `?type`, or repeating the lookup for the direct superclass of `?type` if no such declaration exists. (The syntax "`not exists F[x]`" means that there is no value `v` for which `F[x] = v`.)

erence queues are used by Java Virtual Machine to invoke `finalize` methods.

- More sophisticated reflection analysis. For example, Doop uses distinct representations of instances of `java.lang.Class` for every class in the analyzed program. This reduces the number of human configuration points, solves more reflection scenarios automatically, and improves precision.
- More precise class initialization, modeling better the Java Language Specification.
- More precise handling of cast and assignment compatibility checking.
- More precise exception analysis, using logic that is mutually recursive with the main points-to logic. Exceptions are propagated over the context-sensitive call graph, caught exceptions are filtered, and the order of exception handlers is considered. In a separate publication [3] we describe on-the-fly exception analysis in detail and demonstrate its impact on precision, especially for object-sensitive analyses. On-the-fly exception analysis is expressible highly elegantly in Doop—another benefit of the declarative specification approach.
- Native methods are simulated in a more principled way. In Paddle, indirect method calls via native code are sometimes not represented explicitly, but shortcut directly from the Java call to the Java method. We model the call graph edges more precisely, which is important if applications need a correct call graph.

The declarative approach was of great help in adding language feature support. A major benefit is that semantic extensions are well localized and do not affect the basic definitions (e.g., those in Section 3.2) at all. In contrast, several features in the Paddle framework (e.g., privileged actions, finalization, threads) have their implementation span multiple components.

A second advantage of the declarative approach is that the logic is high-level and often very close to the Java Language Specification. A striking example is the implementation of the logic for the Java cast checking—i.e., the answer to the question "can type A be cast to type B?" Figure 3 shows the full logic, directly from the Doop implementation, with the text of the Java Language Specification in the comments preceding each rule. As can be seen, the Datalog code is almost an exact transcription of the Java specification. (The main difference is that the specification is written in a *must* style, whereas the Datalog code specifies which casts *may* happen. The "must" property is ensured by the least-fixpoint evaluation of Datalog.)

```
/** If S is an ordinary (nonarray) class, then:
 *     o If T is a class type, then S must be the
 *       same class as T, or a subclass of T.
 */
CheckCast(?s, ?s) <- ClassType(?s).
CheckCast(?s, ?t) <- Subclass(?t, ?s).

/**    o If T is an interface type, then S must
 *       implement interface T.
 */
CheckCast(?s, ?t) <- ClassType(?s),
                     Superinterface(?t, ?s).

/** If S is an interface type, then:
 *     o If T is a class type, then T must be Object
 */
CheckCast(?s, "java.lang.Object") <- InterfaceType(?s).

/**    o If T is an interface type, then T must be the
 *       same interface as S or a superinterface of S
 */
CheckCast(?s, ?s) <- InterfaceType(?s).
CheckCast(?s, ?t) <- InterfaceType(?s),
                     Superinterface(?t, ?s).

/** If S is a class representing the array type SC[],
 *  that is, an array of components of type SC, then:
 *     o If T is a class type, then T must be Object.
 */
CheckCast(?s, "java.lang.Object") <- ArrayType(?s).

/**    o If T is an array type TC[], that is, an
 *       array of components of type TC, then one
 *       of the following must be true:
 *          + TC and SC are the same primitive type
 */
CheckCast(?s, ?t) <- ArrayType(?s), ArrayType(?t),
                     ComponentType(?s, ?sc),
                     ComponentType(?t, ?sc),
                     PrimitiveType(?sc).

/**         + TC and SC are reference types (2.4.6),
 *            and type SC can be cast to TC by
 *            recursive application of these rules.
 */
CheckCast(?s, ?t) <- ComponentType(?s, ?sc),
                     ComponentType(?t, ?tc),
                     ReferenceType(?sc),
                     ReferenceType(?tc),
                     CheckCast(?sc, ?tc).

/**    o If T is an interface type, T must be one of
 *       the interfaces implemented by arrays (2.15).
 */
CheckCast(?s, "java.lang.Cloneable") <- ArrayType(?s).
CheckCast(?s, "java.io.Serializable") <- ArrayType(?s).
```

Figure 3. Checkcast implementation in Doop.

3.4 Discussion

Doop currently supports a rich range of analyses with standard precision enhancements from the research literature. This range includes or exceeds practically all precise context-sensitive analyses demonstrated to be feasible in prior literature. We refer throughout the paper to the precision characteristics of the analyses in Doop, especially by reference to other systems. In order, however, to classify the Doop-supported analyses in the larger spectrum of pointer

analysis mechanisms, it is convenient to explicitly list the major features for completeness:

- DOOP implements *subset-based* (or *inclusion-based*) analyses, which preserve the directionality of assignments (unlike *equivalence-based* analyses).
- There is fully on-the-fly callgraph discovery. Additionally, the propagation of analysis facts is limited to reachable methods (i.e., takes the callgraph into account).
- The analyses are *field-sensitive*, which distinguishes between the different fields of an object (as opposed to "field-insensitive"), and between fields of different objects (as opposed to "field-based").
- The analyses can have different kinds of context-sensitivity (call-site, thread- or object-sensitivity) as well as a context-sensitive heap abstraction ("heap cloning"). The context of a called method can be chosen from the current context as well as the context of the receiver object.
- The analyses are array-element insensitive, i.e. elements of an array are not distinguished.
- The analyses take type information into account: points-to facts are not propagated if they would violate the JVM type system.
- DOOP integrates several specialized precision enhancements. For instance, a straightforward but imprecise way to model the flow of the receiver object in virtual method dispatch is by an assignment of the base *variable* of the virtual call (`?base` in Figure 1) to `this`. This is imprecise, since the same virtual method call can invoke different methods, depending on the type of the receiver object. These methods all receive the same points-to set for `this` if the base variable is assigned to `this`. Instead, we combine the assignment of receiver objects with virtual method dispatch and assign a *specific* receiver object (`?heap` in Figure 1) to `this`. This precision improvement is borrowed from PADDLE.
- DOOP only considers special methods (constructors, private, and superclass methods) reachable if the base variable of the invocation points to any objects. Unlike virtual method invocations, the target of a special method invocation does not depend on the run-time class of the object. Therefore, it is tempting to ignore the objects the base variable points to. However, if the variable does not point to any objects, then the method cannot be invoked. This precision improvement is borrowed from PADDLE as well.
- Just as in the PADDLE framework, DOOP can achieve some of the benefits of flow-sensitivity for local variables, by applying the analysis on the static single assignment (SSA) form of the program, e.g. the SSA variant of Soot's Jimple intermediate representation of Java bytecode.

The above list immediately serves to classify the DOOP-supported analyses as much more precise and full-featured than previous declarative pointer analyses in the literature. Specifically, the bddbddb system [28, 29] lacks in support for many Java features, such as native code, reflection, finalization, etc., whose handling constitutes a large part of the DOOP analyses. Although sophisticated client analyses have been implemented on top of bddbddb (e.g., jchord [21]) these analyses are such that they can tolerate unsound handling of Java features, and they act as pure clients: their sophistication does not benefit in any way the precision of the base points-to analysis. Similarly, the quite sophisticated reflection analysis of Livshits et al. [19] is expressed on top of bddbddb's points-to analysis, but is not strictly declarative since it depends on facts computed by a Java pre-analysis, and only applies to context-insensitive analyses. (As mentioned earlier, context-sensitivity in bddbddb is cloning-based and, thus, relies on having a pre-computed call-graph. Integrating this with reflection would be non-trivial.) Furthermore, the reflection analysis of Livshits et al. produces an incorrect call-graph, because it does not take into account the possibility of dynamic dispatch for methods invoked reflectively. This observation is perhaps indicative of the difference between treating language features as an integral part of a declarative points-to analysis intended as the basis for sound inferences, vs. separating the base points-to analysis from language feature support. DOOP's handling of reflection can be viewed as analogous to adding a sophisticated analysis similar to Livshits et al.'s, but in conjunction with a context-sensitive points-to analysis, to obtain the full benefit from the mutual increase in precision of both component analyses.

To illustrate the gap in analysis sophistication between bddbddb and DOOP (as well as PADDLE), we performed the same context-insensitive analysis in both frameworks for the DaCapo benchmark programs. (DaCapo v.2006-10-MR2, JDK 1.4–j2re1.4.2_18, bddbddb svn revision 654, joeq compiler framework revision 2483.) Compared to DOOP, bddbddb reports roughly half the reachable methods (max: 74%, min: 17%, median: 53%, over the 10 DaCapo applications) and less than one-quarter of the points-to facts (max: 64%, min: 3%, median: 21%). The discrepancy is due entirely to the incompleteness of the points-to logic in bddbddb, since the analyses have the same inherent precision. (Increased precision would be unlikely to account for such a dramatic reduction in reachable methods anyway: even the most precise, highly context-sensitive analysis in the DOOP and PADDLE set barely reduces the number of reachable methods by 3-4%.)

In the past, researchers have questioned whether it is even possible to express purely declaratively a full-featured points-to analysis (comparable to PADDLE, which uses imperative code with support for relations [17]). Lhoták [15] writes:

"[E]ncoding all the details of a complicated program analysis problem (such as the interrelated analyses [on-the-fly call graph construction, handling of Java features]) purely in terms of subset constraints [i.e., Datalog] may be difficult or impossible."

Doop demonstrates that an elegant declarative specification is possible and even easy.

Although Doop is a flexible framework, it is not suited to all kinds of analyses. A clear limitation, for instance, is that the context-depth used in the analysis has to be bounded. Doop cannot support analyses that keep an unbounded number of calling contexts, even if the number is guaranteed to be finite (e.g., recursive cycles are flattened). This is due to the lack of constructors/functions in Datalog. This observation is unlikely to have any bearing in practice, however, since other precision enhancements, such as a context-sensitive heap, have been shown to be a better trade-off than an unbounded number of contexts [18]. Combining a context-sensitive heap with even small bounds in context sensitivity (e.g., 4-context-sensitive) is sufficient to make an analysis explode in complexity.

4. Illustration of Doop Optimizations

A declarative specification has advantages in terms of modularity, ease of understanding, and conciseness of expression. One more advantage, however, is that it decouples the analysis logic from its implementation, and allows high-level reasoning about implementation choices. In Doop we have used a novel optimization methodology to convert initial specifications into highly efficient algorithms. Because Doop is expressed in a version of Datalog that exposes indexing decisions to the language level, we can illustrate the optimizations as just Datalog program transformations.

We begin with some background information on Datalog runtimes and the particular engine we use.

4.1 Background: Efficient Datalog Evaluation

A standard optimization for Datalog (indeed, a virtual prerequisite for high performance implementations) is the *semi-naive* evaluation strategy. Semi-naive evaluation keeps track of relation "deltas" on every recursive step, which correspond to the new facts produced by the step. In this way, the next step's results are derived incrementally by using only the previous step's deltas, in all their possible join combinations with full relations. Consider the evaluation of the example from Section 2, reproduced below:

```
1 VarPointsTo(?var, ?heap) <-
2    AssignHeapAllocation(?var, ?heap).
3 VarPointsTo(?to, ?heap) <-
4    Assign(?from, ?to), VarPointsTo(?from, ?heap).
```

Initially, relation `VarPointsTo` is empty. The first step populates relation `VarPointsTo` with the facts from `AssignHeapAllocation`, as dictated by lines 1-2. The rule in lines 3-4 has nothing to contribute, since `VarPointsTo` was empty at the beginning of the step. In the second step, however, this rule joins the new members of `VarPointsTo` from step 1, $\Delta\texttt{VarPointsTo}_1$, with those of input relation `Assign`. This produces $\Delta\texttt{VarPointsTo}_2$, i.e., the new members of `VarPointsTo` from step 2. The next step only needs to join $\Delta\texttt{VarPointsTo}_2$ with `Assign`, in order to produce $\Delta\texttt{VarPointsTo}_3$, and so on.

This optimization is straightforward, yet crucial. It is a major benefit that we get for free from using a declarative language for specifying our analysis. There are more benefits that Doop receives for free through standard Datalog implementation techniques. Specifically, local join optimization is performed: a good order of joins in a single Datalog rule is automatically determined based on statistics on the size of relations and selectivity of joins. This baseline is valuable but still leaves us orders of magnitude away from the performance of a state-of-the-art context-sensitive program analysis. For this we need optimizations across rules, introduction of new database indexes, etc. These optimizations are typically not well-automatable: they correspond to producing an efficient algorithm from a specification, and require human intervention.

In order to execute Datalog programs efficiently, the low-level representation of relations should be compact and an indexing scheme should be in place so that all rules are executed efficiently. The LogicBlox Datalog engine used for Doop allows the user to specify maximum cardinalities for the domains of variables (e.g., the maximum number of values for `?var` in relation `VarPointsTo(?var, ?heap)`). These are used to store domain values as integers and all values of variables (*keys*) in the same relation (`?var` and `?heap` in our example) are packed together in the smallest number of machine words possible using bit shifts and mask operations. A relation is then represented as a sequence of these packed integers for which the relation is true. (Alternatively, the user can specify that the default value for the relation is "false", in which case the system stores all packed keys for which the relation is false. So far we have not used this capability in Doop because all points-to results are very sparse relations.)

As in all database languages, efficiency of execution typically depends on what indexes are defined on the data so that relational operations can be highly efficient. A unique feature of the Datalog engine that we use is that the indexing is exposed to the Datalog language level. In this way, introducing and eliminating indexes can be viewed as just a program transformation, instead of needing to edit the data schema or other configuration files. Specifically, a relation, e.g., `VarPointsTo(?var, ?heap)`, is stored with its contents (pairs of packed variable values) ordered by *innermost* variable, i.e., `?heap`, and then by the next innermost variable, i.e., `?var`, etc. The relation is indexed using a B-tree with a key consisting of all variables together. Since, however, a B-tree is an ordered map, knowing the value of the innermost variable alone is sufficient for efficient indexing. (I.e., the in-

nermost variable is the major index, the second innermost is the next major index, etc.) Thus, variable ordering is very important. The user can change the indexing efficiency to optimize joins, by just reordering variables. For instance, *a join between two relations is very fast if both relations have the join variables in their innermost positions and in the same order*. In that case, both relations just need to be traversed linearly and their contents merged. Another scheme for an efficient join is when joining over the innermost variable of one relation and the second relation is small (so it can be iterated exhaustively and bind the index variable of the first relation). As a rule of thumb, when a relation is known to be small, the local query optimizer will automatically choose to perform the join by iterating exhaustively over its contents. The iteration will bind variables of other relations being joined. These variables should be in the innermost positions, so that their values can be used for efficient indexing. Our optimization methodology, described next, exploits this technique, in particular considering semi-naive evaluation.

In summary, the use of Datalog in Doop separates the specification of an analysis from its implementation, therefore allowing multiple techniques for efficient execution, all expressed at the level of Datalog evaluation. Our current Datalog engine is in many ways mature, but only uses very simple data structures (B-trees and an explicit representation of relations). It is tempting in the future to consider alternative Datalog execution techniques (e.g., the option to transparently use BDDs to represent relations) especially if these are provided in a well-engineered implementation.

4.2 Optimization Methodology

Based on this understanding of Datalog evaluation and optimization opportunities, we next present the optimization techniques we use in Doop through examples.

Consider a refinement of our above rudimentary two-rule pointer analysis logic. We will add to our analysis *field sensitivity*: heap objects can be stored to and loaded from instance fields and the analysis keeps track of such actions. (This example ignores other language features such as method calls—i.e., we assume the analyzed program is just a single `main` function.) Two new input relations are derived from the code of a Java program: `LoadInstanceField(?base, ?signature, ?to)` and `StoreInstanceField(?from, ?base, ?signature)`. The former tracks a load from the object referenced by variable `?base` in the field identified by `?signature`. If, for instance, the Java program contains an action "`x = v.fld;`", then `LoadInstanceField` contains an entry with `?base` being the representation of Java variable "v", `?signature` identifying field "`fld`", and `?to` corresponding to "x". `StoreInstanceField` tracks store actions in a similar manner: Every Java program action "`v.fld = u;`" corresponds to an entry in `StoreInstanceField(?from, ?base, ?signature)`, with `v` represented by variable `?base`, `u` represented by `?from`, and an identifier for field `fld` captured by `?signature`.

Our simple analysis can then be elaborated: (The first two rules are the same but two more rules are added.) A new relation, `InstanceFieldPointsTo`, is used to compute which heap object (`?baseheap`) can point to which other (`?heap`) through a given field (`?signature`).

```
1  VarPointsTo(?var, ?heap) <-
2     AssignHeapAllocation(?var, ?heap).
3  VarPointsTo(?to, ?heap) <-
4     Assign(?from, ?to), VarPointsTo(?from, ?heap).
5  VarPointsTo(?to, ?heap) <-
6     LoadInstanceField(?base, ?signature, ?to),
7     VarPointsTo(?base, ?baseheap),
8     InstanceFieldPointsTo(?baseheap, ?signature, ?heap).
9
10 InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
11    StoreInstanceField(?from, ?base, ?signature),
12    VarPointsTo(?base, ?baseheap),
13    VarPointsTo(?from, ?heap).
```

Reordering Transformation. The above is a straightforward way to express the analysis, but the resulting program is highly inefficient. (Recall that the order of variables in the above relations reflects how the relations are indexed.) In particular, the joins of line 4, 6-8, and 11-13 are all costly. In line 4, neither relation has the join variable in its innermost position. In particular, relation `VarPointsTo` is recursive. After the first step, Datalog's semi-naive evaluation will only need to join the delta of the `VarPointsTo` relation (i.e., a small relation) to produce the new results for the next step. Therefore, it makes sense to reorder the variables of relation `Assign` so that it is indexed efficiently based on variable bindings produced by `VarPointsTo`. That is, the program will be more efficient if relation `Assign` is stored as `Assign(?to, ?from)` rather than `Assign(?from, ?to)`, because variable `?from` is bound by iterating over the contents of small relation Δ`VarPointsTo`. (Of course, this decision on how to store `Assign` may adversely affect joins in other parts of the program—we will soon see how to resolve this.) Similar observations apply to the joins in lines 6-8 and 11-13: no relation has a join variable in its innermost position. Just by applying simple reorderings we can produce a much more efficient implementation:

```
1  VarPointsTo(?heap, ?var) <-
2     AssignHeapAllocation(?heap, ?var).
3  VarPointsTo(?heap, ?to) <-
4     Assign(?to, ?from), VarPointsTo(?heap, ?from).
5  VarPointsTo(?heap, ?to) <-
6     LoadInstanceField(?to, ?signature, ?base),
7     VarPointsTo(?baseheap, ?base),
8     InstanceFieldPointsTo(?heap, ?signature, ?baseheap).
9
10 InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
11    StoreInstanceField(?from, ?signature, ?base),
12    VarPointsTo(?baseheap, ?base),
13    VarPointsTo(?heap, ?from).
```

Folding Transformation. The idea we used in the above transformation is general. *The key novel principle of our op-*

timization methodology is that, for highly recursive Datalog programs (such as our points-to analyses), the primary determinant of performance is whether the relation deltas produced by semi-naive evaluation bind all the variables needed to index into other relations. In this way, exhaustive traversal of non-deltas is avoided. To achieve this effect, we often need to introduce new indexes. Since in our Datalog engine an index is always tied to the order of relation variables, to obtain a new index we need to introduce new relations. This is done through applications of the *folding* program transformation [5]. Folding introduces a temporary relation that holds the result of intermediate joins. This can improve performance in many ways. First, it can re-order variables in the intermediate relation and, thus, introduce a new index, so that further joins are more efficient. Second, it can cache intermediate results, implementing the "view materialization" database optimization. Third, it can be used to guide the query optimizer to perform joins between smaller relations first, so as to minimize intermediate results. Finally, it can be used to project out unnecessary variables, thus keeping intermediate results small.

Many of these benefits can be obtained in our simple pointer analysis program. Consider the 3-way join in lines 11-13 of the above "optimized" program. Since relation `VarPointsTo` is recursive and used twice, either of its instances can be thought of as a "small" relation from the perspective of join efficiency. Specifically, under semi-naive evaluation, one can think of the above rule (in lines 10-13) as equivalent to the following delta-rewritten program:

```
ΔInstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
   StoreInstanceField(?from, ?signature, ?base),
   ΔVarPointsTo(?baseheap, ?base),
   VarPointsTo(?heap, ?from).
ΔInstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
   StoreInstanceField(?from, ?signature, ?base),
   VarPointsTo(?baseheap, ?base),
   ΔVarPointsTo(?heap, ?from).
```

(We elide version numbers, since we are just making an efficiency point. Note that the deltas are also part of the full relation—i.e., they are the deltas from the previous step. Hence, we do not need a third rule that joins two deltas together.)

The first rule is fairly efficient as-is: the delta relation binds variable `?base`, which is used to index into relation `StoreInstanceField` and bind variable `?from`, which is used to index into relation `VarPointsTo(?heap, ?from)`. The second rule, however, would be disastrous if executed as-is: none of the large relations has its innermost variable bound by the delta relation. We could improve the performance of the second rule by reordering the variables of `StoreInstanceField` but there is no way to do so without destroying the performance of the first rule.

This conflict can be resolved by a fold. We introduce a temporary relation that captures the result of a two-relation join, projects away unnecessary variables, and reorders the remaining variables so that the join with the third relation is highly efficient. This results in the following optimized program, with intermediate relation `StoreHeapInstanceField` introduced.

```
1  VarPointsTo(?heap, ?var) <-
2     AssignHeapAllocation(?heap, ?var).
3  VarPointsTo(?heap, ?to) <-
4     Assign(?to, ?from), VarPointsTo(?heap, ?from).
5  VarPointsTo(?heap, ?to) <-
6     LoadInstanceField(?to, ?signature, ?base),
7     VarPointsTo(?baseheap, ?base),
8     InstanceFieldPointsTo(?heap, ?signature, ?baseheap).
9
10 InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
11    StoreHeapInstanceField(?baseheap, ?signature, ?from),
12    VarPointsTo(?heap, ?from).
13
14 StoreHeapInstanceField(?baseheap, ?signature, ?from) <-
15    StoreInstanceField(?from, ?signature, ?base),
16    VarPointsTo(?baseheap, ?base).
```

Note that the last two rules only contain relations with the same innermost variables, therefore any delta-execution of those rules is efficient. Implicitly, this is achieved because the folding also adds a new index, for the new intermediate relation.

The above program still admits more optimization, as one more inefficient join remains. Consider the joins in lines 6-8 of the above program. Both relation `VarPointsTo` and relation `InstanceFieldsPointsTo` are recursively defined. (There is direct recursion in `VarPointsTo`, as well as mutual recursion between them.) Thus, after the first step, their deltas will be joined with the full other relations. Specifically, in semi-naive evaluation the above rule (lines 5-8) is roughly equivalent to:

```
ΔVarPointsTo(?heap, ?to) <-
   LoadInstanceField(?to,?signature,?base),
   ΔVarPointsTo(?baseheap,?base),
   InstanceFieldPointsTo(?heap, ?signature, ?baseheap).
ΔVarPointsTo(?heap, ?to) <-
   LoadInstanceField(?to, ?signature, ?base),
   VarPointsTo(?baseheap, ?base),
   ΔInstanceFieldPointsTo(?heap, ?signature, ?baseheap).
```

As before, the performance problem is with the second delta rule: the innermost variable of the large relations is not bound by the delta relation. It is tempting to try to eliminate the inefficiency with a different variable order, without performing more folds. Indeed, we could optimize the joins in lines 3-8 without an extra fold, by reordering the variables of `VarPointsTo` as well as `LoadInstanceField`—the latter so that `?signature` is last. This would conflict with the joins in lines 10-16, however, and would require further rewrites.

Therefore, the inefficiency can be resolved with a fold, which will also reorder variables so that all joins are highly efficient: the joined relations always have a common innermost variable. We introduce the intermediate relation

LoadHeapInstanceField, and get our final highly-optimized program:

```
1  VarPointsTo(?heap, ?var) <-
2     AssignHeapAllocation(?heap, ?var).
3  VarPointsTo(?heap, ?to) <-
4     Assign(?to, ?from), VarPointsTo(?heap, ?from).
5  VarPointsTo(?heap, ?to) <-
6     LoadHeapInstanceField(?to, ?signature, ?baseheap),
7     InstanceFieldPointsTo(?heap, ?signature, ?baseheap).
8
9  LoadHeapInstanceField(?to, ?signature, ?baseheap) <-
10    LoadInstanceField(?to, ?signature, ?base),
11    VarPointsTo(?baseheap, ?base).
12
13 InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
14    StoreHeapInstanceField(?baseheap, ?signature, ?from),
15    VarPointsTo(?heap, ?from).
16
17 StoreHeapInstanceField(?baseheap, ?signature, ?from) <-
18    StoreInstanceField(?from, ?signature, ?base),
19    VarPointsTo(?baseheap, ?base).
```

Programmer Insights. Note that the above optimization decisions are intuitively appealing, although no intuition was used in deriving them. For instance, a programmer with an understanding of the domain will likely prefer this ordering of variables in VarPointsTo. (Recall that the innermost variable yields the most important indexing with our B-tree ordering.) The relation seems intuitively much more useful when treated as a map of program variables to heap objects, rather than as a map of heap objects to variables that can point to them. Values flow through variables in a points-to analysis, not through heap objects directly.

Additionally, the introduction of temporary relation LoadHeapInstanceField orders the three-way join of LoadInstanceField, VarPointsTo, and InstanceFieldPointsTo so that the first two relations are joined first. This is good, since LoadInstanceField is likely smaller than InstanceFieldPointsTo: the former is an input relation, with its contents in one-to-one correspondence with a subset of program instructions, while the latter is inferred from a subset of program instructions joined with the (multiply recursive) points-to relation, resulting in a transitive closure computation.

Such insights can sometimes guide the optimization effort, but they are just heuristics. In the end, we have not found dramatic performance differences between optimization paths that both end up with joins that are syntactically efficient, i.e., have the join variables in innermost positions and always bound by a recursive relation so that its delta is used. This syntactic criterion is, more than anything else, the primary determinant of performance.

Impact. Perhaps surprisingly, the above compact set of optimizations and insights are the main source of the efficiency of Doop, compared to a naive Datalog implementation. Applying these optimizations on a full pointer analysis for realistic programs results in improvements of over 3 orders of magnitude: run-time is often dropped from many hours to mere seconds. Furthermore, the optimizations are robust with respect to the different analysis variants supported in Doop. The same optimized trunk of code is used for analyses with several different kinds of context sensitivity.

5. Doop Performance

We next present performance experiments for Doop, and especially contrast it with Paddle—a BDD-based framework that is state-of-the-art in terms of features and scalability. Because of the variety of experimental results, a roadmap is useful:

- We first evaluate Doop in "Paddle-compatibility" mode. In this mode, Doop results are precisely equivalent to Paddle. This, however, means that the analysis does not support exceptions, which Doop treats very differently. In this mode, Doop is much faster (6.6x to 16.3x in median speedup) than Paddle for standard context-sensitive analyses (1-call-site, 1-call-site+heap, 1-object, 1-object+heap).
- We then compare the full analyses of Doop with the full Paddle. The Doop analyses are not exactly equivalent, but are strictly *more* precise than their Paddle counterparts. In this "full-mode", Doop outperforms Paddle by 10x for call-site-sensitive analyses (including heavier ones, such as 1-call-site+heap) scales similarly or better than Paddle for even the heaviest object-sensitive analyses in Paddle's experiment set, and even handles analyses that Paddle does not, such as a 2-call-site-sensitive and a 2-object-sensitive analysis, both with a context-sensitive heap.
- Finally, we discuss the lessons learned from comparing an explicit representation approach with a BDD-based one. We see that the performance discrepancy between Doop and Paddle is well-explained when one considers the total size of BDDs for the call-graph, var-points-to, and field-points-to relations. The numbers cast doubt on whether BDDs can be the best representation of relations in analyses similar to the ones we have considered.

Preliminaries and Experimental Setup. We use a 64-bit machine with two quad-core Xeon 2.33GHz CPUs (only one thread was active at a time, except for Paddle runs, where the Java garbage collector ran on a different thread). The machine has 16GB of RAM and 4MB of L2 cache (actually 8MB of L2 cache per CPU, but every 2 cores share 4MB). (For comparison, the Paddle study [18] was conducted on a fairly comparable 4-way 2.6GHz Opteron machine, also with 16GB of RAM. Although we do not compare absolute numbers with that study, it is useful for context to know that qualitative scalability estimates are not due to hardware discrepancies.)

We analyzed the DaCapo benchmark programs, v.2006-10-MR2, with JDK 1.4 (j2re1.4.2_18), which is much larger than JDK 1.3 used (with the same programs) by Lhoták and Hendren [18]. Since recent points-to analysis algorithms (e.g., [10, 11]) claim scaling to "a million lines of code", we

should point out that our benchmarks are the largest in the literature on *context-sensitive* points-to analysis.

We contacted PADDLE's author Ondřej Lhoták to confirm input parameters for optimal performance (including optimal BDD variable orderings). The initial settings of the analysis are identical to those in the most recent PADDLE study [18]. PADDLE takes an option for an initial number of BDD nodes to allocate, which can be used to reduce garbage collection. We do not use this option for several reasons. *1)* This initial number is also the maximum number of nodes, which requires knowing up-front how complex an analysis will be for a specific benchmark. *2)* Setting this number to the maximum required value would immediately consume all virtual memory, independent of the specific benchmark. For comparison, we want memory consumption to be proportional to the complexity of an analysis. *3)* The performance benefit of setting an initial number of BDD nodes was limited in our experiments (less than 10%), and does not change any conclusions.

When referring to different analyses we use the prefixes "N-call-site-sensitive", "N-call-site", or just "N-call" for an N-call-site-sensitive analysis, and "N-object-sensitive" or just "N-object" for an N-object-sensitive analysis, as well as the suffixes "+N-heap" or just "+NH" for an analysis with a context-sensitive heap with N (object or call-site) contexts kept. (We omit the N if it can only be 1.) E.g., "2-call+1H" designates a 2-call-site-sensitive analysis with a context-sensitive heap using 1 call-site as context for heap object abstractions; "1-call+H" designates a 1-call-site-sensitive analysis with a context-sensitive heap (which can only have 1 call-site as context).

We consider any analysis that takes more than 7200 seconds (2 hours) to have failed.

The software, benchmark scripts, and more statistics are available at `http://doop.program-analysis.org/oopsla09`.

5.1 PADDLE-Compatibility

We first evaluate DOOP and PADDLE in a mode in which the results are equivalent. We worked hard to ensure semantic equivalence to a high degree. All operations on relations are designed to be logically equivalent. That is, all propagation of facts and all intermediate relations are virtually identical.

Comparing the results of pointer analyses is challenging because of many minor differences between the analyses. Also, minor differences frequently propagate everywhere, making it difficult to locate the source of an issue. Nevertheless, we achieved *exact* equivalence of reachable methods, reachable method contexts, context-sensitive call graph edges, instance field points-to, static field points-to, and variable points-to information. We compared the results automatically and report any differences. The various improvements of DOOP over PADDLE in support for Java language features (Section 3.3) have been patched in Paddle and submitted as bug reports (e.g., reference object support), or disabled in DOOP (e.g., reflection analysis) for this comparison.

There is one algorithmic enhancement applied to DOOP as well as PADDLE in PADDLE-compatibility mode: DOOP treats static initializer methods (`clinit`) context-insensitively. Static initializers are not affected by any context, so they can be treated context-insensitively for all of the context abstractions we study. (For very different kinds of analyses, e.g., a thread-sensitive analysis, this will not be true.) This enhancement (as well as other logical enhancements discussed later) is not significant for PADDLE (it strictly improves performance but only marginally), because PADDLE can avoid redundancy through its use of BDDs. The enhancement is, however, important for DOOP's explicit representation of relations.

Notably, for the experiments in PADDLE-compatibility mode, both DOOP and PADDLE ignore control- and data-flow induced by exceptions. This is necessary, since the DOOP handling of exceptions is significantly different from PADDLE's.

Figures 4 to 8 display the execution times of DOOP vs. PADDLE for five analyses, ranging from context-insensitive to 1-object-sensitive+heap. As can be seen, DOOP is an order of magnitude faster than PADDLE for the context-insensitive analysis (min: 7.4x, max: 10.9x, median: 10x), the 1-call-site-sensitive analysis (min: 7.9x, max: 19.6x, median: 16.3x), and the 1-object-sensitive analysis (min: 3.2x, max: 18.1x, median: 15.2x). For the heavier analyses, DOOP is almost always significantly faster than PADDLE, except for the `bloat` benchmark and a 1-object-sensitive+heap analysis. Specifically, DOOP exhibits a median speedup of 7.3x for the 1-call-site-sensitive+heap analysis (min: 1.8x, max: 9.2x) and a median speedup of 6.6x for the 1-object-sensitive+heap analysis (min: 0.9x, max: 7.3x). (Recall that the latter is the analysis that Lhoták and Hendren considered to require a research breakthrough to implement efficiently without BDDs.)

The analysis times in seconds illustrate the significance of the speedup: for most programs, analysis time is dropped from several hundreds of seconds to just a few tens of seconds.

Generally, DOOP in PADDLE-compatibility mode scales very well even to much more complex analyses (e.g., 2-object+heap). Nevertheless, recall that the PADDLE-compatibility mode does not support Java exception handling. Adding exception handling in a way that is compatible with PADDLE would artificially distort DOOP performance. PADDLE exception handling is highly imprecise, treating every exception throw as an assignment to a single global variable. The variable is then read at the site of an exception catch. This approach ignores the information about what exceptions can propagate to a catch site: all catch sites become related with all type-compatible throw sites and with each other. This very approximate treatment affects the precision of the analysis results but barely affects performance for PADDLE: the BDD representation of relations tolerates the

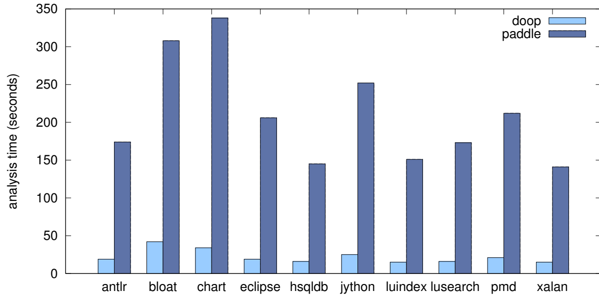

Figure 4. (Paddle-compatibility mode) context-insensitive

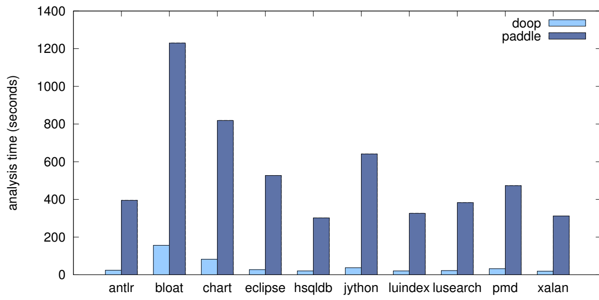

Figure 5. (Paddle-compatibility mode) 1-call

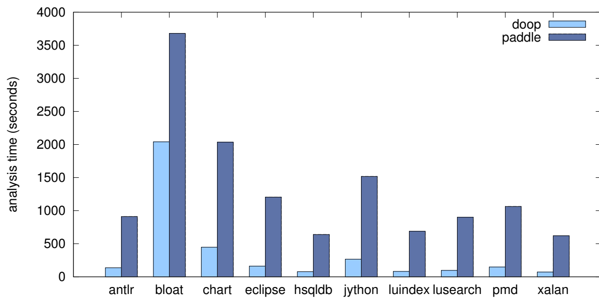

Figure 6. (Paddle-compatibility mode) 1-call+H

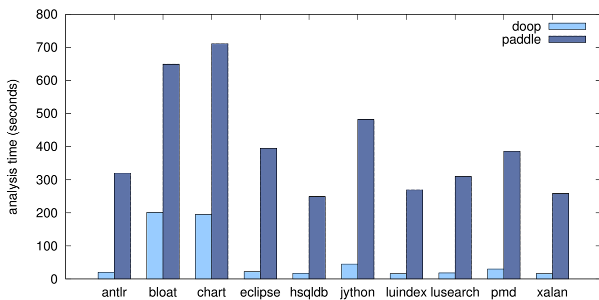

Figure 7. (Paddle-compatibility mode) 1-object

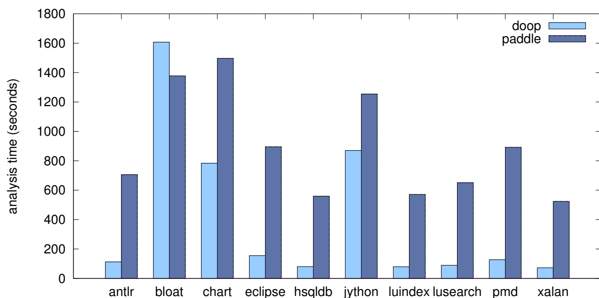

Figure 8. (Paddle-compatibility mode) 1-object+H

redundancy in the computed relations (e.g., a higher number of facts in the call-graph or the var-points-to relations) since the extra facts are highly regular. The BDD representations of relations in Paddle are hardly larger, even with significant exception-handling-induced imprecision. In contrast, Doop's explicit representation of relations cannot tolerate the addition of such "regular" imprecision without suffering performance penalties. This phenomenon is perhaps counter-intuitive: Doop performs much better when imprecision is avoided, which is also a desirable feature for the quality of the analysis.

5.2 Full Doop Performance and Precision

Our main experimental results compare the full version of Doop with the full Paddle, and present detailed statistics on the precision of Doop analyses.

The full mode of Doop is not exactly equivalent to the full Paddle, yet the Doop analysis logic is always strictly more precise and more complete, resulting in higher-quality analyses. The differences are in the more precise and complete handling of reflection, more precise handling of exceptions, etc.

Figures 9 to 16 compare the performance of Doop and Paddle. (The analyses presented are a representative selection for space and layout reasons.) This range of analyses reproduces the most demanding analyses in Lhoták and Hendren's experiment set [18] and includes analyses that even exceed the capabilities of Paddle: 2-call+1-heap, 2-object+1-heap, and 2-call+2-heap. As can be seen, Doop is often significantly faster, especially for call-site-sensitive analyses (e.g., a large speedup for 1-call-site—min: 5.0x, max: 12.9x, median: 9.7x—and for 1-call-site+heap—min: 2.3x, max: 16.7x, median: 12.3x).

Doop is not as fast for object-sensitive analyses, but recall that it performs a much more precise analysis than Paddle because of its precise exception handling. On-the-fly exception handling results in a dramatic, 2x increase in var points-to precision (i.e., on average each variable is inferred to point to half as many objects) for object-sensitive analyses [3]. Still, Doop outperforms Paddle for the vast majority of data points, even for the heaviest analyses in the Paddle set. For the 1-object+heap analysis Doop is faster for 8 out of 10 benchmarks (min: 0.4x, max: 4.0x, median: 3.0x). The only benchmark for which Doop is significantly slower is `xalan`, but this outlier is due to Paddle's less complete *reflection* analysis. Paddle misses a large part of the call graph (only reports 3722 reachable methods, instead of 6468 reported by Doop) and analyzes much less code.

The significance of these results cannot be overstated: The conventional wisdom has been that such analyses cannot be performed without BDDs. For instance, Lhoták and Hendren write regarding the Paddle study: *"It is the use of BDDs and the* Paddle *framework that finally makes this study possible. Moreover, some of the characteristics of the analysis results that we are interested in would be very costly*

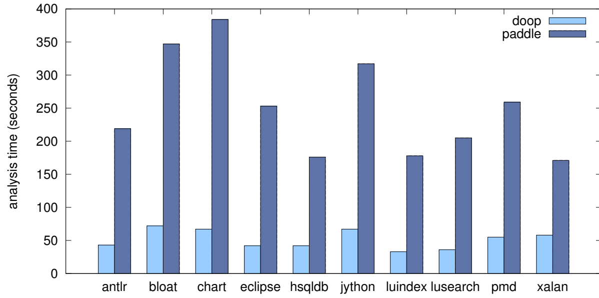

Figure 9. (Full mode) context-insensitive

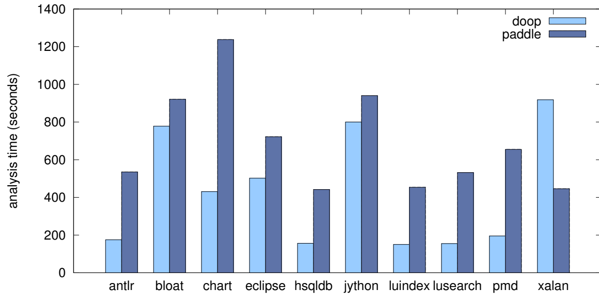

Figure 10. (Full mode) 1-object

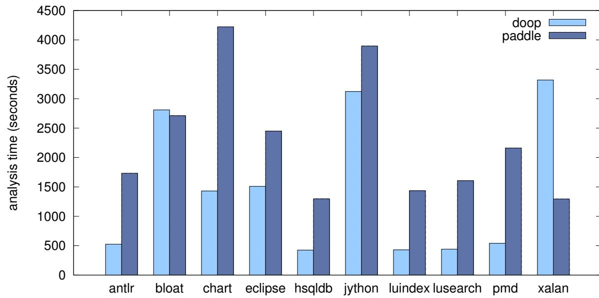

Figure 11. (Full mode) 1-object+H

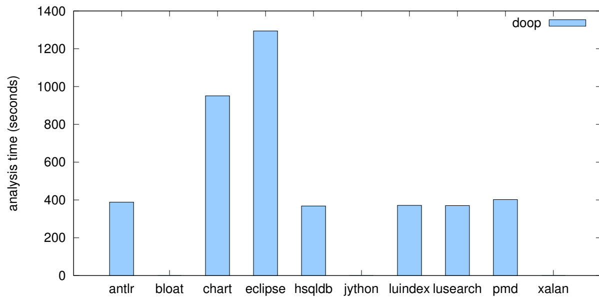

Figure 12. (Full mode) 2-object+1H

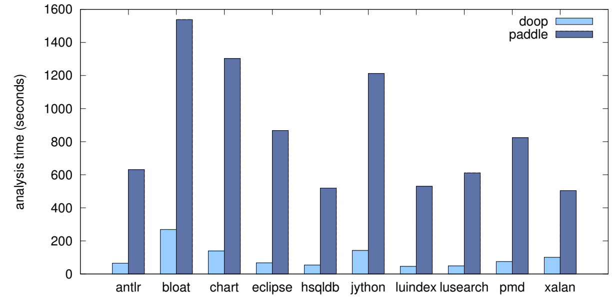

Figure 13. (Full mode) 1-call

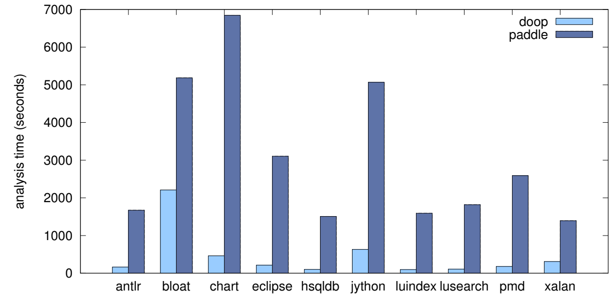

Figure 14. (Full mode) 1-call+H

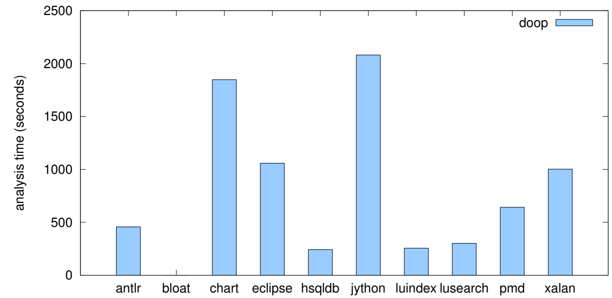

Figure 15. (Full mode) 2-call+1H

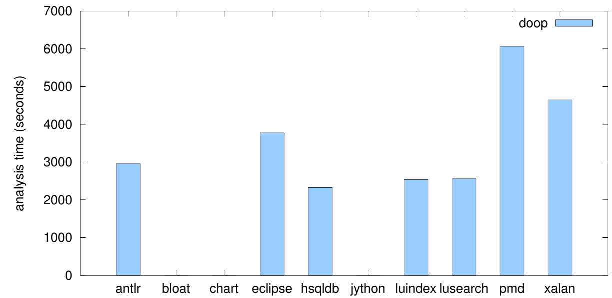

Figure 16. (Full mode) 2-call+2H

to measure on an explicit representation." [18] (Recall also that the PADDLE study analyzed the DaCapo benchmarks with the smaller JDK 1.3.1_01.)

The last three analyses of our set (2-call+1-heap, 2-object+1-heap, and 2-call+2-heap) are more precise than any context-sensitive analyses ever reported in the research literature. With a time limit of 2 hours, DOOP analyzed most of the DaCapo applications under these analyses. All three analyses are impossible with PADDLE. The first two are not supported by the PADDLE framework, while the third is too heavy to run. (In our tests, the analysis times-out even for the smallest of the DaCapo benchmarks. Lhoták also reports that "*[He] never managed to get* PADDLE *to run in available memory with these settings*".[2])

The range of DOOP-supported analyses allows us to obtain insights regarding analysis precision. Figure 17 shows some

[2] http://www.sable.mcgill.ca/pipermail/soot-list/2006-March/000601.html

of the most important statistics on our analyses' results for representative programs. Perhaps the most informative metric is the average points-to set size for plain program variables.[3] The precision observations are very similar to those in the PADDLE study: object-sensitivity is very good for ensuring points-to precision, and a context-sensitive heap can only serve to significantly enhance the quality of results. We can immediately see the value of our highly precise analyses, and especially the combination of a 2-object-sensitive analysis with a context-sensitive heap. This most precise analysis typically drops the average points-to set size to one-tenth of the size of the least precise (context insensitive) analysis. Remarkably, this even impacts the number of call-graph edges—a metric that notoriously improves very little with increasing the precision of the points-to analysis. In future work we expect to conduct a thorough evaluation of the precision of a wide range of analyses for several end-user metrics.

5.3 BDDs vs. Explicit Representation

Generally, the performance differences between DOOP and PADDLE are largely attributable to the use of BDDs vs. an explicit representation of relation contents. The comparison of the two systems reveals interesting lessons regarding the representation of relations in points-to analysis.

BDDs are a maximally reduced data structure (for a given variable ordering) so they naturally trade off some time for space, at least for large relations. Furthermore, BDDs have heavy overheads, in the case of irregular relations that cannot be reduced. Consider the worst-case scenario for BDDs: a relation with a single tuple. The BDD representation in PADDLE uses a node per bit—e.g., the single tuple in a relation over a 48-bit variable space will be represented by 48 BDD nodes. Each node in the BuDDy library (used by PADDLE) is 20 bytes, or 160 bits. This represents a space overhead of 160x, but it also represents a time overhead, since what would be a very quick operation in an explicit tuple representation now requires traversing 48 heap objects (allocated in a single large region, but with no structure-locality).

The difficulty in analyzing the trade-off is that results on smaller data sets and operations do not translate to larger ones. For instance, we tried a simple experiment to compare the join performance of DOOP and PADDLE, without any other recursion or iteration. We read into memory two previously computed points-to analysis relations (including the `VarPointsTo` relation, for which PADDLE's BDD variable order is highly optimized) and computed their join. The fully expanded relation size in DOOP was a little over 1GB, or 7 million tuples. DOOP performed the join in 24.4 seconds.

[3] Note the apparent paradox of having the average number of var-points-to facts often be higher when computed over context-sensitive variables than over plain variables. Although each context-sensitive variable has fewer points-to facts than its context-insensitive version, the average over all context-sensitive variables can be higher: program variables that have many points-to facts are also used in many more contexts, skewing the results.

	analysis	nodes	edges	var points-to			
antlr	insens	4510	24K	2.8M	67	-	-
	1-call	4498	24K	897K	22	4.9M	31
	1-call+H	4495	24K	887K	22	14M	90
	2-call+1H	4484	23K	719K	18	48M	84
	2-call+2H	4451	23K	570K	14	79M	171
	1-obj	4486	24K	748K	18	4.7M	16
	1-obj+H	4435	23K	435K	11	25M	86
	2-obj+1H	4382	22K	264K	7	7.8M	8
chart	insens	7873	41K	5.9M	84	-	-
	1-call	7820	40K	2.6M	36	18M	66
	1-call+H	7816	40K	2.5M	36	43M	162
	2-call+1H	7800	40K	2.2M	31	202M	173
	2-call+2H	×	×	×	×	×	×
	1-obj	7803	40K	2.4M	34	18M	27
	1-obj+H	7676	37K	1.2M	17	81M	123
	2-obj+1H	7570	35K	414K	6	24M	7
pmd	insens	5536	27K	3.5M	73	-	-
	1-call	5519	26K	1.1M	22	5.8M	31
	1-call+H	5516	26K	1.0M	22	16M	89
	2-call+1H	5506	26K	925K	20	65M	94
	2-call+2H	5473	25K	803K	17	136M	219
	1-obj	5504	26K	964K	21	5.2M	15
	1-obj+H	5440	25K	682K	15	25M	77
	2-obj+1H	5372	24K	302K	7	7.4M	7
xalan	insens	6580	33K	3.4M	62	-	-
	1-call	6568	33K	1.4M	25	7.5M	35
	1-call+H	6565	33K	1.4M	25	22M	104
	2-call+1H	6551	32K	1.2M	22	78M	88
	2-call+2H	6505	32K	939K	17	125M	170
	1-obj	6549	33K	1.2M	22	19M	30
	1-obj+H	6468	31K	696K	13	106M	173
	2-obj+1H	×	×	×	×	×	×

Figure 17. Precision statistics of DOOP analyses for a subset of the DaCapo benchmarks. The columns show call-graph nodes and edges, as well as total and average (per variable) points-to facts, first for plain program variables and then for "context-sensitive variables" (i.e., context-variable tuples).

PADDLE spent 40x more time, 957 seconds, creating the BDD, but then performed the join in just 0.527 seconds. In terms of space, the BDD representation of the 7 million tuples consisted of just 148.7 thousand nodes—less than 3MB of memory! This demonstrates how different the cost model is for the two systems. If PADDLE can exploit regularity and build a new BDD through efficient operations on older ones, then its performance is unparalleled. Creating the BDD, however, can often be extremely time consuming. Furthermore, a single non-reducible relation can become a bottleneck for the whole system. Thus, it is hard to translate the results of microbenchmarks to more complex settings, as the complexity of BDDs depends on their redundancy.

To gain a better understanding of performance, we analyzed the sizes of BDDs in PADDLE for some major relations in its analyses, relative to the size of the explicit representations of the same relations. Figure 18 shows the sizes of rela-

tions "nodes" (representing the context-sensitive call-graph nodes, i.e., context-qualified reachable methods), "edges" (i.e., context-sensitive call-graph edges), var points-to (the main points-to relation, for context-qualified vars), and field points-to (the points-to relation for object fields). For each relation, the table shows the size of its explicit representation (measured in number of rows—i.e., number of total facts in the relation), the size of the BDD representation (in number of BDD nodes) and the ratio of these two numbers—although they are in different units the variation of the ratios is highly informative.

The above numbers are for Paddle as configured for our Paddle-compatibility experiments, so that the BDD statistics can be directly correlated to the performance of Doop (explicit representation) vs. Paddle (BDDs). Examination of the table in comparison with Figures 4-8 reveals that the performance of Paddle relative to Doop is highly correlated with the overall effectiveness of BDDs for relation representation. For benchmarks and analyses for which Paddle performs better compared to Doop, we find that all four relations (or at least the largest ones, if their size dominates the sizes of others) exhibit a much lower ratio of BDD-nodes-to-facts than in other benchmarks or analyses. Consider, for instance, the 1-object+heap analysis. The BDD size statistics reveal that `bloat` and `jython` are significant outliers compared to the rest of the DaCapo applications: their BDD-nodes-to-facts ratios are much lower for all large relations. A quick comparison with Figure 8 reveals that Paddle performs unusually well for these two benchmarks.

This understanding of the performance model for the BDD-based representation leads to further insights. The ultimate question we want to answer is whether (and under what conditions) there is enough regularity in relations involved in points-to analyses for BDDs to be the best representation choice. *Figure 18 suggests that this is not the case, at least for the analyses studied here.* The main way to improve the performance of the BDD representation is by changing the BDD variable ordering. The BDD variable ordering used in our Paddle experiments is one that minimizes the size of the var points-to relation (which, indeed, consistently has a small BDD-nodes-to-facts ratio in Figure 18). This order was observed by Lhoták to yield the best results in terms of performance. (It is worth noting that the Paddle authors were among the first to use BDDs in program analysis, have a long history of experimentation in multiple successive systems, and have experimented extensively with BDD variable orderings until deriving ones that yield "impressive results" [2].) Nevertheless, what we see in Figure 18 is that it is very hard to provide a variable ordering that minimizes all crucial BDDs. Although the var points-to relation is consistently small, the (context-sensitive) call-graph edge relation is inefficient and it is usually large enough to matter. All current techniques utilizing BDDs for points-to analysis (e.g., in bddbddb or Paddle) require BDD variable orderings *"that are simultaneously good for the many BDDs in a system of interrelated analyses"* [15]. It does not, therefore, seem likely that BDDs will be the best representation option for precise context-sensitive points-to analyses without significant progress in our understanding of how BDDs can be employed.

6. Related and Future Work

Fast and Precise Pointer Analysis. There is an immense body of work on pointer analysis, so we need to restrict our discussion to some representative and recent work. Fast and precise pointer analysis is, unfortunately, still a trade-off. This is unlikely to change. Most recent work in pointer analysis explores methods to improve performance by reducing precision strategically. The challenge is to limit the loss of precision, yet gain considerably in performance. For instance, Lattner et al. show [14] that an analysis with a context-sensitive heap abstraction can be very efficient by sacrificing precision using unification constraints. This is a common sacrifice. Furthermore, there are still considerable improvements possible in solving the constraints of the classic inclusion-based pointer analysis of Andersen, as illustrated by Hardekopf and Lin [10].

In full context-sensitive pointer analysis, there is an ongoing search for context abstractions that provide precise pointer information, and do not cause massive redundant computation. Milanova suggested that an object-sensitive analysis [20] is an effective context abstraction for object-oriented programs, which was confirmed by Lhoták's extensive evaluation [18]. Several researchers have argued for the benefits of using a context-sensitive heap abstraction to improve precision [18, 22].

The use of BDDs attempts to solve the problem of the large amount of data in context-sensitive pointer analysis by representing its redundancy efficiently [2, 29]. The redundancy should ideally be eliminated by choosing the right context abstraction. Xu and Rountev's recent work [30] addresses this problem. Their method aims to determine context abstractions that will yield the same points-to information. This is an exciting research direction, orthogonal to our work on declarative specifications and optimization. However, in their specific implementation, memory consumption is growing quickly for bigger benchmarks, even on Java 1.3.

IBM Research's Wala [7] static analysis library is designed to support different pointer analysis configurations, but no results of Wala's accuracy or speed have been reported in the literature. It will be interesting to compare our analyses to Wala in future work.

Reflection and Program Analysis. Reflection, dynamic class loading, and native methods are a major issue for static program analysis. Paddle inherits support for many native methods from its predecessor, Spark [16]. Paddle's support for reflection is relatively unsophisticated compared to the reflection analysis of Livshits specified in Datalog on top

		call-graph nodes			call-graph edges			var points-to			field points-to		
		facts	bdd	ratio	facts	bdd	ratio	facts	bdd	ratio	facts	bdd	ratio
context-insensitive	antlr	4K	1K	0.35	23K	95K	4.23	2.0M	58K	0.03	766K	28K	0.04
	bloat	6K	2K	0.26	46K	132K	2.86	7.9M	81K	0.01	1.0M	38K	0.04
	chart	8K	3K	0.35	39K	163K	4.19	5.3M	101K	0.02	1.8M	51K	0.03
	eclipse	5K	2K	0.34	24K	104K	4.39	2.4M	63K	0.03	746K	31K	0.04
	hsqldb	4K	1K	0.41	17K	80K	4.71	1.5M	50K	0.03	493K	23K	0.05
	jython	6K	2K	0.31	32K	123K	3.90	3.3M	72K	0.02	750K	34K	0.04
	luindex	4K	1K	0.38	18K	86K	4.70	1.5M	53K	0.03	567K	25K	0.04
	lusearch	4K	2K	0.34	21K	98K	4.65	1.8M	59K	0.03	606K	28K	0.05
	pmd	5K	2K	0.32	25K	113K	4.51	2.5M	62K	0.02	652K	28K	0.04
	xalan	4K	1K	0.40	17K	80K	4.78	1.4M	50K	0.04	501K	23K	0.05
1-call-site-sensitive	antlr	22K	37K	1.64	83K	682K	8.26	2.9M	735K	0.26	636K	28K	0.04
	bloat	45K	55K	1.21	266K	1.1M	4.32	30M	1.5M	0.05	792K	39K	0.05
	chart	39K	64K	1.67	164K	1.2M	7.09	18M	1.6M	0.09	1.4M	52K	0.04
	eclipse	23K	38K	1.64	113K	705K	6.22	4.0M	852K	0.21	572K	32K	0.06
	hsqldb	17K	29K	1.73	61K	523K	8.62	2.1M	590K	0.28	395K	24K	0.06
	jython	31K	47K	1.51	139K	907K	6.53	5.7M	1.0M	0.18	539K	35K	0.06
	luindex	18K	31K	1.73	65K	559K	8.63	2.4M	645K	0.27	459K	26K	0.06
	lusearch	21K	36K	1.69	76K	638K	8.41	2.9M	751K	0.26	488K	29K	0.06
	pmd	25K	42K	1.69	94K	769K	8.14	4.7M	843K	0.18	512K	29K	0.06
	xalan	17K	29K	1.74	60K	519K	8.64	2.1M	595K	0.29	396K	24K	0.06
1-call-site-sensitive+heap	antlr	22K	37K	1.63	83K	682K	8.26	8.9M	2.4M	0.27	12M	7.3M	0.59
	bloat	45K	55K	1.22	251K	1.1M	4.55	159M	7.3M	0.05	27M	10M	0.38
	chart	39K	64K	1.66	164K	1.2M	7.11	42M	6.3M	0.15	26M	16M	0.63
	eclipse	23K	38K	1.64	113K	706K	6.23	14M	3.1M	0.23	9.4M	7.1M	0.75
	hsqldb	17K	29K	1.73	61K	523K	8.61	6.2M	1.8M	0.30	5.7M	4.3M	0.76
	jython	31K	47K	1.50	139K	908K	6.54	22M	4.2M	0.19	15M	8.6M	0.58
	luindex	18K	31K	1.73	65K	560K	8.63	7.0M	2.1M	0.30	6.4M	5.0M	0.78
	lusearch	21K	36K	1.70	76K	637K	8.40	8.5M	2.5M	0.30	7.8M	5.7M	0.74
	pmd	25K	42K	1.69	94K	768K	8.13	14M	3.1M	0.22	8.2M	6.7M	0.82
	xalan	17K	29K	1.74	60K	518K	8.64	6.1M	1.8M	0.30	5.7M	4.3M	0.77
1-object-sensitive	antlr	36K	19K	0.54	218K	489K	2.25	1.5M	324K	0.22	25K	33K	1.33
	bloat	71K	27K	0.38	1.8M	1.2M	0.65	14M	646K	0.05	307K	44K	0.14
	chart	81K	38K	0.47	1.0M	1.1M	1.14	16M	763K	0.05	60K	58K	0.97
	eclipse	40K	22K	0.55	312K	596K	1.91	1.9M	381K	0.20	27K	36K	1.33
	hsqldb	31K	17K	0.55	170K	412K	2.43	1.1M	271K	0.25	17K	28K	1.69
	jython	64K	26K	0.40	746K	742K	0.99	4.9M	455K	0.09	38K	39K	1.02
	luindex	32K	18K	0.57	178K	436K	2.44	1.2M	294K	0.24	18K	30K	1.73
	lusearch	35K	20K	0.57	202K	492K	2.43	1.5M	335K	0.23	20K	34K	1.71
	pmd	42K	21K	0.50	309K	557K	1.80	2.6M	373K	0.14	40K	34K	0.85
	xalan	30K	17K	0.56	168K	411K	2.45	1.1M	274K	0.25	16K	28K	1.73
1-object-sensitive+heap	antlr	35K	19K	0.55	161K	448K	2.79	8.6M	797K	0.09	2.3M	505K	0.22
	bloat	69K	27K	0.39	1.4M	1.0M	0.73	56M	1.9M	0.03	13M	1.2M	0.09
	chart	76K	37K	0.49	647K	973K	1.50	41M	1.9M	0.05	9.1M	1.3M	0.14
	eclipse	39K	22K	0.56	212K	544K	2.56	11M	1.0M	0.10	2.8M	631K	0.23
	hsqldb	30K	17K	0.56	131K	380K	2.90	6.3M	656K	0.10	1.7M	409K	0.24
	jython	62K	25K	0.41	638K	684K	1.07	76M	1.4M	0.02	15M	1.1M	0.07
	luindex	31K	18K	0.58	134K	402K	2.99	6.4M	695K	0.11	1.7M	427K	0.26
	lusearch	34K	20K	0.58	147K	447K	3.04	7.3M	785K	0.11	1.8M	488K	0.26
	pmd	41K	21K	0.52	216K	499K	2.31	10M	892K	0.09	2.8M	539K	0.19
	xalan	30K	17K	0.57	129K	379K	2.93	6.0M	665K	0.11	1.5M	411K	0.27

Figure 18. BDD statistics for the most important context-sensitive relations of Paddle: total number of facts in the context-sensitive relation, number of BDD nodes used to represent those facts, and the ratio of BDD nodes / total number of facts.

of Whaley's bddbddb [19]. In particular, PADDLE does not maintain information about `Class` objects created through `Class.forName`, which requires very conservative assumptions about later `Class.newInstance` invocations. However, the reflection analysis of Livshits was only integrated in a context-insensitive pointer analysis. The fully declarative nature of DOOP allows us to use very similar Datalog rules also in context-sensitive analyses.

Declarative Programming Analysis. Program analysis using logic programming has a long history (e.g., [4, 23]), but this early work only considers very small programs. In recent years, there have been efforts to apply declarative program analysis to much larger codebases and more complex analysis problems. We discussed the relation to Whaley's work on context-sensitive pointer analysis using Datalog and BDDs [29] throughout this paper. The DIMPLE [1] analysis framework has shown to be competitive in performance for *context-insensitive* pointer analysis using tabled Prolog. The demonstrated pointer analysis of DIMPLE uses a conservative, pre-computed call graph, so the analysis is reduced to propagation of points-to information of assignments, which can be very efficient. DOOP expresses all the logic of a *context-sensitive* pointer analysis in Datalog.

Demand-Driven and Incremental Analysis. A demand-driven evaluation strategy reduces the cost of an analysis by computing only those results that are necessary for a client program analysis [12, 26, 27, 31]. This is a useful approach for client analyses that focus on specific locations in a program, but if the client needs results from the entire program, then demand-driven analysis is typically slower than an exhaustive pointer analysis. Reps [24] showed how to use the standard magic-sets optimization to automatically derive a demand-driven analysis from an exhaustive analysis (like ours). This optimization combines the benefits of top-down and bottom-up evaluation of logic programs by adding side-conditions to rules that limit the computation to just the required data.

More recently, Saha and Ramakrishnan [25] explored the application of incremental logic program evaluation strategies to context-insensitive pointer analysis. As pointed out in this work, the algorithms for materialized view maintenance and incremental program analysis are highly related. As we discussed, incremental evaluation is also crucial for DOOP's performance. The large number of reachable methods in an empty Java program[4] suggests that incremental analysis could bring down the from-scratch evaluation time substantially. We have not explored these incremental evaluation scenarios yet. The engine we use also supports incremental evaluation after deletion and updates of facts using the DRed [8] algorithm. Efficient incremental evaluation might make context-sensitive pointer analysis suitable for use in IDEs.

[4] Even an empty Java program causes the execution of a number of methods from the standard library. This causes a static analysis to compute an even larger number of reachable methods, especially when no assumptions are made about the loading environment (e.g., security settings and where the empty class will be loaded from).

7. Conclusions

We presented DOOP: a purely declarative points-to analysis framework that raises the bar for precise context-sensitive analyses. DOOP is elegant, full-featured, modular, and high-level, yet achieves remarkable performance due to a novel optimization methodology focused on highly recursive Datalog programs. DOOP uses an explicit representation of relations and cha(lle)nges the community's understanding on how to implement efficient points-to analyses.

Acknowledgments This work was funded by the NSF (CCF-0917774, CCF-0934631) and by LogicBlox Inc. We thank Ondřej Lhoták for his advice on benchmarking PADDLE, Oege de Moor and Molham Aref for useful discussions, the anonymous reviewers for helpful comments, and the LogicBlox developers for their practical help and support.

References

[1] W. C. Benton and C. N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *PPDP '07: Proc. of the 9th ACM SIGPLAN int. conf. on Principles and practice of declarative programming*, pages 13–24, New York, NY, USA, 2007. ACM.

[2] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI*, pages 103–114. ACM, 2003.

[3] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In L. Dillon, editor, *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2009. To appear.

[4] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *PLDI '96: Proc. of the ACM SIGPLAN 1996 conf. on Programming language design and implementation*, pages 117–126, New York, NY, USA, 1996. ACM.

[5] S. K. Debray. Unfold/fold transformations and loop optimization of logic programs. In *PLDI '88: Proc. of the ACM SIGPLAN 1988 conf. on Programming Language design and Implementation*, pages 297–307, New York, NY, USA, 1988. ACM.

[6] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proc. of the 30th int. conf. on Software engineering*, pages 391–400, New York, NY, USA, 2008. ACM.

[7] S. J. Fink. T.J. Watson libraries for analysis (WALA). `http://wala.sourceforge.net`.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proc. of the 1993 ACM SIGMOD int. conf. on Management of data*, pages 157–166, New York, NY, USA, 1993. ACM.

[9] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 2–27. Spinger, 2006.

[10] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI'07: Proc. ACM SIGPLAN conf. on Programming Language Design and Implementation*, pages 290–299, New York, NY, USA, 2007. ACM.

[11] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 226–238, New York, NY, USA, 2009. ACM.

[12] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI '01: Proc. of the ACM SIGPLAN 2001 conf. on Programming language design and implementation*, pages 24–34, New York, NY, USA, 2001. ACM.

[13] 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 *PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[14] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Not.*, 42(6):278–289, 2007.

[15] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.

[16] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th Int. Conf.*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[17] O. Lhoták and L. Hendren. Jedd: a bdd-based relational extension of java. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 158–169, New York, NY, USA, 2004. ACM.

[18] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008.

[19] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780. Springer-Verlag, Nov. 2005.

[20] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[21] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 308–319, 2006.

[22] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Importance of heap specialization in pointer analysis. In *PASTE '04: Proc. of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2004. ACM.

[23] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

[24] T. W. Reps. Solving demand versions of interprocedural analysis problems. In *CC '94: Proc. of the 5th Int. Conf. on Compiler Construction*, pages 389–403, London, UK, 1994. Springer-Verlag.

[25] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP '05: Proc. of the 7th ACM SIGPLAN int. conf. on Principles and practice of declarative programming*, pages 117–128, New York, NY, USA, 2005. ACM.

[26] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI '06: Proc. of the 2006 ACM SIGPLAN conf. on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM.

[27] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *OOPSLA '05: Proc. of the 20th annual ACM SIGPLAN conf. on Object oriented programming, systems, languages, and applications*, pages 59–76, New York, NY, USA, 2005. ACM.

[28] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[29] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.

[30] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08: Proc. of the 2008 int. symposium on Software testing and analysis*, pages 225–236, New York, NY, USA, 2008. ACM.

[31] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *POPL '08: Proc. of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, New York, NY, USA, 2008. ACM.