NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Incremental Static Analysis with Differential Datalog

Argyro A. Ritsogianni

Supervisor (or supervisors): **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Ph.D Researcher NKUA

ATHENS

OCTOBER 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σταδιακή Στατική Ανάλυση με την Διαφορική Datalog

Αργυρώ Α. Ριτσογιάννη

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Διδάκτορας Ερευνητής ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2019

BSc THESIS

Incremental Static Analysis with Differential Datalog

Argyro A. Ritsogianni
S.N.: 1115201400171

SUPERVISOR: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Ph.D Researcher NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σταδιακή Στατική Ανάλυση με την Διαφορική Datalog

Αργυρώ Α. Ριτσογιάννη
Α.Μ.: 1115201400171

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Διδάκτορας Ερευνητής ΕΚΠΑ

ABSTRACT

Many applications have their code updated by several maintenance transformations throughout the application's functioning lifetime. Therefore, the results of analyzing an application may need to be evaluated incrementally. In this thesis, we explore the possibilities of incrementality in static program analysis, using the Doop framework and the DDlog incremental Datalog engine. Doop is a static analysis framework and DDlog (Differential Datalog) is an engine for incremental Datalog evaluation, based on a data-parallel library, Differential Dataflow.

We find that Doop-based static analyses can be incrementally evaluated via DDlog requiring minimum interventions to the analysis logic. We illustrate DDlog's performance compared to the Soufflé Datalog engine that Doop integrates.

SUBJECT AREA: Incremental static program analysis

KEYWORDS: static program analysis, differential datalog, incrementality,

differential dataflow

ΠΕΡΙΛΗΨΗ

Πολλές εφαρμογές ενημερώνουν τον κώδικα τους με αρκετούς μετασχηματισμούς συντήρησης καθ 'όλη τη διάρκεια ζωής της εφαρμογής. Επομένως, τα αποτελέσματα της ανάλυσης μιας εφαρμογής μπορεί να χρειαστεί να αξιολογηθούν σταδιακά. Στην παρούσα πτυχιακή, διερευνούμε τις δυνατότητες σταδιακής αύξησης της στατικής ανάλυσης προγράμματος, χρησιμοποιώντας τη βιβλιοθήκη Doop και τη μηχανή Datalog της DDlog. Το Doop είναι ένα στατικό πλαίσιο ανάλυσης και η DDlog (Differential Datalog) είναι ένας μηχανισμός για αυξητική αξιολόγηση Datalog, βασισμένη σε μια βιβλιοθήκη παραλληλισμού δεδομένων, τη Differential Dataflow.

Διαπιστώνουμε ότι οι στατικές αναλύσεις που βασίζονται σε Doop μπορούν να αξιολογηθούν αυξητικά μέσω της DDlog, η οποία απαιτεί ελάχιστες παρεμβάσεις στη λογική ανάλυσης. Παρουσιάζουμε την απόδοση της DDlog σε σύγκριση με το μηχανισμό Soufflé Datalog που το Doop ενσωματώνει.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αυξητική/Σταδιακή στατική ανάλυση προγράμματος

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση προγράμματος, διαφορική datalog,

αύξηση, διαφορική dataflow

To my family.

ACKNOWLEDGMENTS

I would like to thank my prof. Yannis Smaragdakis for suggesting the idea of my thesis and for giving me the opportunity to work on such an interesting topic.

I would also like to thank my supervisor, PhD researcher George Fourtounis, for the continuous support and the help towards completing this thesis by providing help and suggestions throughout it, and every PLaST Lab member, for sharing their knowledge.

October 2019

CONTENTS

LIST OF FIGURES

PREFACE

This project, which was developed as my undergraduate thesis in the University of Athens, Greece between February 2019 and October 2019, aims to investigate the incremental prospects of the Differential Datalog engine and whether the implementations of this engine can get close to the optimized Soufflé one, using some of the minimal analyses contained in Doop (“micro” and “self-contained”).

1. INTRODUCTION

Programs are developed incrementally and undergo several maintenance transformations during their lifetime. Therefore, when statically analyzing a program, it is desirable to evaluate the analysis logic incrementally, while the program code changes. We investigate the promise of incremental static program analysis in the context of the Doop framework and DDlog engine.

Doop is a static analysis framework that performs a collection of various analyses, especially for Java programs, declaratively using the Datalog language.

DDlog (Differential Datalog) [3] is an incremental Datalog engine, based on the differential dataflow library. Differential dataflow is an efficient data-parallel framework for large data, quickly updating data views on changes. By using DDlog, we can obtain incremental evaluation of Doop-based static analyses, with minimal changes to the analysis logic.

We explain the changes that are required to be performed to port a part of Doop's static analyses' logic to DDlog, the conversions that are needed and how we could improve on them.

We investigate the performance of DDlog in comparison to Doop's default Datalog engine Soufflé. We discuss the amount of time and memory consumed in case of non-incremental execution. We then present the results of DDlog's incremental implementation times for representative programs.

Eventually, we find how feasible incrementality is in static analysis and thus introduce the possibility of expanding the static analysis for incremental execution, opening the way for future experimentation.

2. BACKGROUND

2.1 Doop Framework and Pointer Analysis

In the program analysis domain, the declarative expression of analyses, for adaptability and clearer specification, is desirable. Datalog[15] is a popular query language that is used for these kinds of analyses. Datalog is simultaneously logic with declarative semantics and a realistic programming language, which adapts the syntax style of Prolog, defining recursive relations. Computation in Datalog includes monotonic logical inferences that iteratively apply to generate more facts until a fix-point is reached.

Doop is a static analysis framework for Java byte-code which encodes multiple pointer or points-to analysis algorithms by using Datalog. Doop has comprehensive analysis algorithms that produce important information on a variety of existing static analysis techniques [2].

Various implementations of program analysis are based on Datalog, both high-level and low-level. High-level implementations are more abstract ones like a mathematical formalism and low-level implementations more systematic and specific ones. Doop's approach uses the Datalog language for describing static program analysis, a domain full of mutually recursive definition sets, each managing a specific language feature.

Formerly, Doop used a commercial Datalog engine, created by LogicBloxInc but Doop presently uses Soufflé, a open-source Datalog engine, which translates a Datalog program to a parallel C++ program. Soufflé [4, 7] is a variant of Datalog aiming to support tool designers that exercise skill in making static analyses. Soufflé is based on advanced compilation techniques for logic programs and can achieve benefits of high performance evaluation.

Doop's main framework for inputting java programs [8] for declarative processing is Soot. It accepts as input a Java program's byte-code and produces input facts to be inserted into the database so analysis rules can process them. Consequently, there is no need for source code for the framework to execute the analysis, thus closed-source libraries are also capable to be analyzed. In Datalog terminology, EDB (Extensional Database) predicates are actually the input data of a Datalog program. The contents of these predicates are created during the derivation of new facts by the Soot program preprocessing step.

Static analysis intends to automatically extract a program's properties from the program code. Pointer analysis [1] is a foundational static program analysis, having a wide variety of applications and an abundant literature. The purpose of pointer analysis is to provide, for a pointer variable or expression an estimation of the group of program objects that the variable or expression can refer to.

A static model of the main structure for global program data, the heap, is based on information on the values of pointer variables or expressions. Pointer analysis computes such information inter-procedurally, offering a convenient substrate for further analysis algorithms.

A simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation, once the preprocessing step is finished, in the following specification:

VarPointsTo(?heap, ?var) :- AssignHeapAllocation(?heap,?var).

VarPointsTo(?heap, ?to) :- Assign(?to, ?from), VarPointsTo(?heap, ?from)

The two simple Datalog rules of the figure above are known as IDB (Intentional Database) rules and can recursively compute and derive new predicates. The first rule above establishes the computation's base and declares that upon an allocated heap object's assignment to a variable, this variable may point-to the heap object. The second recursive rule expresses that, upon a variable's assignment to another, the latter may point-to any heap object that the former variable points-to.

Base algorithms of pointer analysis are hard to incrementalize. Nevertheless, there are approaches that consider the specific needs of a client or program point and automatically regulate the precision to attain significant scalability benefits.

2.2 Differential Dataflow

Differential dataflow [5] is an efficient, fast data-parallel programming framework. It aims to immediately react to arbitrary changes in input collections and to efficiently process a large amount of data. Differential dataflow programs are written to transform collections of data using known operators such as map, filter, join and group-by. Differential dataflow enables the repeated application of dataflow transformations to collections using an iterative operator. Once the differential dataflow computation is established, we can then insert or delete records from the input; the system will automatically inform the output with the proper respective deletions and insertions. Differential dataflow rapidly performs updates when changes in collections appear, because it is based on the timely dataflow computational model for data-parallel programming that automatically parallelizes through multiple computers, processes, and threads.

Incremental updates can boost performance and are efficient and effective for computation and communication in a data-parallel system. What characterizes Differential dataflow is its capability to compute and update arbitrarily nested repetitive data-parallel computations. The general idea is having a computation that retains a collection of differences from which the records can be effectively and efficiently updated instead of having a point place in computation (a dataflow vertex) which preserves a dataset [16].

The quick reaction of Differential Dataflow when a change in input appears gives surprisingly new prospects of scalable computation. Differential dataflow aims to scale in many different directions. It augments the rate at which the user interacts with it and the amount of threads, computers and data it operates on [17, 18].

Differential dataflow programs are written with operations that look a bit like database (SQL) or big data (Map Reduce) idioms against seemingly static input collections. Operators are functions that have to adjust to their input changes and collections are the inputs and outputs of operators. Operators of Differential dataflow programs are applied to collections and end up in collections, to which more and more operations are applied.

For example [5], here is a differential dataflow code in Rust to compute the out-degree distribution of a directed graph:

```
let outdegreedist = edges.map(|(src, dst)| src)    // extract source
                        .count();                  // count  appearances of source
                        .map(|(newsrc, deg)| deg)  // extract degree
                        .count();                  // count appearances of degree
```

In imperative computations [10, 11] it is important to force computations to happen, in contrast to dataflow computations where one has to wait for them to occur. Differential dataflow waits to collect enough information to infer the right answer, thus it will not need much and unnecessary work to execute in the end.

Differential computation can be effectively implemented in the context of a declarative data-parallel dataflow language like Datalog. Incremental computation is a feature which strives to save time when data change occurs by only re-computing the result outputs which are determined by changed data. It is remarkably faster than simply computing

new outputs. This can be a key optimization for program analysis tools expressed in Datalog.

The innovation of differential computation [6] has two parts: first, the computation's state, which differs according to a partially ordered sequence of versions and, second, the set of updates needed to revise the state, which is maintained in an indexed data structure, at any given version, rather than consolidated into a "current" version. Specifically, the state and updates to that state correspond to versions expressed as multi-dimensional timestamps, which permits efficient re-usability. For instance, if version (i, j) matches up with the jth iteration of a loop on the ith round of input, its inference can re-use work done at both former versions (i-1, j) and (i, j-1) in spite of which version was most recently managed by the system. The differential computation model upgrades incremental computation by allowing the existence of various states arranged in a version's partial ordering and retains an index of individual updates, thus to be merged in different ways for different versions.

A dataflow computation [24] is modeled as a directed graph in which vertices correspond to inputs of a program, outputs of a program, or operators (e.g Select, Join, Group-By) and edges connect the use of one vertex's output with the input to another. Generally, many inputs and outputs may exist in a dataflow graph. A dataflow graph may be cyclic but in our case it is necessary for cycles to support fixed-point sub-computations for more efficient total computation.

Interactive Datalog computations can be facilitated by Differential dataflow, both in a way where the source relations can updated arbitrarily and the results will be automatically revised, and top-down queries can be implemented without re-computing the other procedure [19].

Consider, for instance, the technique used in DD for deletions: all tuples produced are kept together with cardinality per derivation's "round". For instance, we have a simple recursive program:

Path(x,y) :- Edge(x,y).

Path(x,y) :- Edge(x,z), Path(z,y).

And for
Edge(1,2), Edge(1,3), Edge(1,4), Edge(4,5), Edge(2,6), Edge(3,6), Edge(5,6)

we get:

round0*: Path(1,2), Path(1,3), Path(1,4), Path(4,5), Path(2,6), Path(3,6), Path(5,6) [specifically, 1xPath(1,2), 1xPath(1,3), etc.]*

round1*: 2xPath(1,6), Path(1,5), Path(4,6)*

round2*: Path(1,6)*

In particular, there is an imposed "breadth-first" ordering of the rules' firings, where every round is over when all of the deltas (over all rules) from the previous semi-naive evaluation round have been evaluated. The DD engine keeps extra information, but not overwhelmingly much: it remembers that Path(1,6) was derived in two different ways in round1, and only in one way in round2, but it does not store exactly "which" were these ways.

If we remove the Edge(1, 2) for an incremental update the result will be:

round0: -Path(1,2) [minus 1xPath(1,2)]
round1: -Path(1,6) [one of the two ways to compute Path(1,6) is now removed]

and no further changes, since nothing else changes in round 1: Path(1,6) is still true, but its inference count drops from 2 to 1, so there is no need for the change to be propagated to round2.

An issue that Differential dataflow may has is the maintenance of computation's trace which sometimes can be as big as or bigger than the computation's output concluding to a large amount of state preservation in memory.

2.3 Differential Datalog

Many real-world applications require revising relatively fast their output in response to input changes.

Consider, for instance, a program analysis framework like Doop which evaluates a set of rules that are defined over the program's intermediate representation. This analyzer can be merged in an IDE to immediately warn the developer for a bug that may appear in the program, thus needing the re-evaluation of the rules after every change. This re-evaluation has to be incremental, maintaining, as much as possible, intermediate results, to accomplish interactivity for large code bases. An incremental algorithm needs to retain for each step intermediate computation results and for each operation needs to carry out an incremental version of it in order to propagate efficiently input changes to the output [23].

A programmer, using DDlog and providing a set of rules to produce output relations relying on input relations, only needs to write an original non-incremental problem in a dialect of Datalog and the DDlog compiler will produce an effective incremental implementation.

Differential dataflow [9] is DDlog's primary execution engine.

DDlog is an incremental Datalog engine which operates in a bottom-up fashion. DDlog is initialized by solid facts that a user provide and while executing Datalog rules produces all possible derived facts, something that top-down engines do not do. DDlog only executes the required and minimum computations to determine all changes in the derived facts, whenever a change to the ground facts happens, thus giving important performance benefits. DDlog extends Datalog with powerful type system, such as common bit-vector and integer arithmetic, simple and expressive functional language, string operations, generic collection types' manipulation and saving, and aggregations' implementation.

DDlog is strongly typed and statically checked language, thus a DDlog program is based on typed relations. DDlog's type system is inspired by Haskell and provides a rich set of types, such as generic types, user-defined types and driven by context types.

Relations in DDlog are consisting of three categories: input, output and intermediate relations. Input relations' content is given by a user or a program in a stepwise fashion. An input relation may have an optional primary key which facilitates the entries' deletion efficiently by specifying it only. DDlog program at runtime will update the environment for changes in output relations while computing these relations. DDlog program computes intermediate relations without informing the environment for any changes.

A labeled directed graph whose vertices symbolize relations is called a dependency graph, which DDlog depends on it. An edge in the graph from relation 'B' to relation 'H' labeled 'p', it implies that there is a rule with relation 'H' in its head and relation 'B' in the body with polarity 'p' negative or positive. We call stratified negation when no cycle in a graph includes an edge indicated with negative polarity. So, DDlog relations are not allowed to recursively based on their own negation, thus recursive rules with stratified negation are permitted to DDlog's programs.

The compiler of DDlog is written in Haskell. The compiler implements various optimization steps, including parsing, type inference and validation. First, the DDlog compiler produces Rust code as text files and then this code is compiled and linked with Differential dataflow's Rust version resulting in native executable's production.

The DDlog compiler's result is a recursive cyclic dataflow graph. Dataflow relational operators compute relations which are symbolized by the dataflow graph's nodes and the input and output DDlog's relations of every operator are connected by the graph's edges. In DDlog the Differential dataflow's implementation of an operator like map, join, aggregation and etc, helps to temporal indexes to trace relations over time enabling efficient incremental evaluation across many cores.

Differential dataflow runtime in DDlog retains temporal indexes that points to previous relations' versions, so as to compute incremental results and decrease memory consumption.

The DDlog's aim in large projects is to minimize the needed conversion between various languages and to improve program's performance and memory usage. Differential Datalog is a Datalog dialect with automated incremental computation for applications' implementations. Although, DDlog's execution process is fully incremental, a programmer does not need to write incremental Datalog programs for DDlog, non-incremental ones are enough.

The DDlog language contains a rich type system, a robust expression language, and a module system. It handles strings and integers and can be inter-operate with several languages.

Compiling DDlog program into a Rust library warrants good performance, but changes to rules or relational schema limits the flexibility demanding thus re-compilation.

A program in DDlog supports two ways of use:

1. The main one, for which also DDlog optimizes, is to be incorporated as library in other applications and the second one is to produce a standalone program which is executed in a command-line fashion for updating and query relation states [20]. A command-line interface program (CLI) is generated by the compiler giving the opportunity for users to directly interact with a DDlog program by inserting or deleting tuples in input relations, starting transactions, committing changes, dumping relations' results and getting statistics for the process time and memory consumption.
2. A transactional API, with bindings available for several languages, is exported to a library, which is produced by the DDlog's compiler for DDlog programs.

Relations in DDlog are sets where identical records cannot exist in the same relation. To filter records and to compute resulting records we can use expressions. An expression is a combination of one or more variables, functions, constants, and operators, which are interpreted and computed to produce another value.

Functions in DDlog compute in a pure and without side-effect manner. Functions that are declared without a body and with extern keyword denote externals functions which are defined outside of a DDlog program. All states needs inside a DDlog function are written in an expression-oriented fashion. At the moment DDlog does not support recursive functions and complicated functions can be carried out as extern ones.

A simple example in DDlog:

```
typedef TType = string

typedef TMethodDescriptor = string

typedef TMethod = string

typedef Tsymbol = TType
```

```
//function that concatenates strings
function cat(s: string, t: string): string = (s ++ t)

output relation Method_Descriptor(_method:TMethod, _descriptor:TMethodDescriptor)
input relation Method_ReturnType(_method:TMethod, _returnType:TType)
input relation Method_ParamTypes(_method:TMethod, _params:Tsymbol)

Method_Descriptor(_method, _descriptor):- Method_ReturnType(_method,
_returnType), Method_ParamTypes(_method, _params), var _descriptor =
cat(_returnType, _params).
```

DDlog provides three ways to insert data to a program: (a) listing statically ground facts (rules without a body) as part of the program, (b) using a text-based command line interface, and (c) from a Rust, C, C++, or Java program. DDlog also offers profiling features to help the programmer examine which parts of a DDlog program occupy the most CPU and memory.

A program in DDlog consists of multiple type definitions, functions, relations and rules. The declarations' ordering does not matter and meta-attributes, which are specifications that determine object's properties, can annotate declarations where appropriate.

An atom in DDlog is a predicate that appears to have positive or negative polarity inside a rule and holds when a relation has given value. The main specification of an atom is listing its fields or giving its value explicitly. A DDlog rule is composed of the head comprised of at least one atom and zero or more body clauses. An atom that is mutually recursive with its head cannot be held in a rule's body which includes an aggregate clause. Variables can be presented in a rule's body clause and are visible in the rule's head and following clauses. Variables have to be referenced in a positive atom using pattern expressions or they have to be strictly in the assignment clause's left-hand side, flat-map or aggregate clause. The aggregate operator which DDlog may use collects records with same values of variables' subset and applies on them an aggregation function.

```
BestPrice(item, best_price) :- Price(.item = item, .price = price), var best_price =
Aggregate((item), group_min(price)).
```

Differential Datalog is based on an enriched Datalog's programming language [21] version and also offers a traditional imperative-language-like syntax to compose relational rules.

3. FROM SOUFFLÉ TO DDLOG

Doop contains two small analyses that we use for our adaptation from Soufflé to DDlog:

1. Doop contains a minimal and filtered, “self-contained” analysis for experimentation with incremental Datalog-like evaluation. This analysis is useful for experimenting with the analysis logic of Doop without having to work with the full rules. The analysis logic is under 150 lines, but there are another ~400 lines of schema, imports, type-based computation e.g. sub-typing and method resolution. The analysis is still not fully realistic e.g. there are lots of dependencies missing, so the application is only partially analyzed, but it is still a good benchmark for showing how Doop’s logic can interact with new features.
2. Doop also contains a “micro” analysis for more realistic implementations. This is a small analysis that interfaces with more parts of the analysis in Doop.

Our goal is to investigate the performance of Differential Datalog vs. Soufflé for Doop analyses. The main question is to see if the incremental implementations can get near to the optimized Soufflé one. DDlog’s syntax is close to that of Soufflé, and there is a parser that mostly automates Soufflé to DDlog conversion so it is fairly easy to run DDlog on Soufflé instances.

Soufflé uses a language that targets rapid-prototyping for analysis problems with logic; enabling deep design-space explorations; designed for large-scale static analysis; e.g. points-to analysis for Java, taint-analysis, and security checks.

The Soufflé converter that exists in DDlog’s Github repository contributes to Datalog’s translation from the Soufflé Datalog dialect to the DDlog dialect. There is also a grammar file which helps parsing Soufflé’s dialect. Soufflé and DDlog have slight differences in their syntax but these differences can be usually translated between the two dialects for Doop’s logic.

Some modifications to the self-contained analysis file are needed. For example, the creation of new auxiliary rules and an alternative way of writing functions and rules, especially those rules that have a negation or disjunction in their body. So as, Soufflé converter can accept this analysis and be able to produce the new DDlog analysis (more easily), without any problem.

The modifications are made by us so as to improve this converter to accept the original analysis with as few changes as possible because some rules are hard to pass and they need some changes to keep up with the strict syntax of DDlog, such as a rule must never begin with a negation.

DDlog is case-sensitive. Relation, constructor, and type variable names have to begin with uppercase letters; function, variable, and argument names must begin with an underscore or lowercase letters. A type name can start with either a lowercase letter or an uppercase letter or underscore. So, in this case DDlog’s soufflé transformer handles all this.

The changes we perform manually are:

- Add support for nested cat function (a recursive function) needing changes in grammar and Soufflé converter file.
- Support generation of only facts simultaneously with Soufflé-to-Ddlog logic conversion for parallel manipulation/execution in Doop framework.
- Use dummy relation in front of assignment because DDlog does not accept negation and assignment as the first relation of rule’s body. For instance:

RHeapAllocation_Type(_heap, _type),

RisHeapAllocation(_heap) :- Rdummy(0), var _heap = string_intern("<<immutable>>"), var _type = string_intern("java.lang.Object").

- Support non-DNF (Disjunctive normal form) conversions in the Soufflé converter, for some disjunction cases that self-contained or micro analysis have and so as to not change the initial code by writing extra rules.

All these changes happen without changing the overall functionality of the Soufflé converter.

Disjunctions in rules are not in DNF form but can be converted by a preprocessor in the Soufflé converter that feeds the converted DNF rules to the rest of the conversion process. We implement this preprocessor as a temporary workaround to experiment more with DDlog, until proper implementation comes. The DNF converter in souffle_converter.py does not currently handle all patterns of DNF conversion. We add it as a short-term solution to support the non-DNF patterns found in Doop, until proper support for full disjunction is added(especially we make all these so we can test micro analysis as well). Basically for this case we add a 'toDNF' flag to souffle_converter.py, to transform non-DNF rules to DNF ones. The transformation is disabled by default and should only support cases that are found in Doop.

For example:

Souffle DNF:

MethodLookup(?simplename, ?descriptor, ?type, ?method) :-

(DirectSuperclass(?type, ?supertype) ; DirectSuperinterface(?type, ?supertype)),

MethodLookup(?simplename, ?descriptor, ?supertype, ?method),

! MethodImplemented(?simplename, ?descriptor, ?type, _).

DDlog non-DNF:

RMethodLookup(_simplename, _descriptor, _type, _method) :- RDirectSuperclass(_type, _supertype), RMethodLookup(_simplename, _descriptor, _supertype, _method), not RMethodImplemented(_simplename, _descriptor, _type, _).

RMethodLookup(_simplename, _descriptor, _type, _method) :- RDirectSuperinterface(_type, _supertype), RMethodLookup(_simplename, _descriptor, _supertype, _method), not RMethodImplemented(_simplename, _descriptor, _type, _).

After completing the translation of Soufflé (Doop) Datalog to Differential Datalog, we can execute it and see its results compared to Soufflé. When DDlog creates the executable which we can execute via the command line interface (CLI) that it offers, then we are able to measure the amount of time for different indicative programs, using as measurement the DDlog's timestamp command for more accuracy.

A remaining difference to different incremental implementations is most likely due to how data are represented internally in DDlog. For example, in many cases this means that a relation may already be in the right format to be joined with another relation. In contrast, DDlog internally generates one big variant type that represents all possible relations and has to un-wrap this type to extract key and value from it, thus possibly having an impact on performance.

The optimization which has the most impact on performance is string interning, a storing method using only one copy of each distinct string value. Also meta-attributes notation for size, improves the performance and the DDlog time. There are other things one

might expect as well in auto-generated code that might have some overheads and may need some optimizations too. These optimizations were implemented by Frank McSherry and Leonid Ryzhyk based on our input benchmarks.

DDlog at this moment supports only one meta-attribute, the size meta-attribute. It is applicable to extern type declarations, defines the corresponding Rust data type's size in bytes and function as indication to compiler so as to optimize efficiently data structure schemes.

Finally we try to perform all the micro analysis with minimal modifications to the original file, especially with negation and disjunctions rules. The results are satisfactory, which gives the opportunity for DDlog's first integration in Doop.

4. RESULTS

At the end of this thesis there are conclusions drawn from the experimental evaluation of the analyses performed by DDlog and Soufflé Doop's default engine.

Our experimental analyses was compiled on Linux OS via DDlog compilation "ddlog –l <program>.dl –action=compile –L lib". First, we performed the self-contained analysis and when all the conversions were made we performed the micro analysis. All the analyses were executed with command "analysisname_cli –w <number> –no-print < in.dat &> dumpfile" and the results were saved in a dump file. The analyses were running on a machine with 200GB free RAM and 32 hardware threads, thus there was no need for swapping, causing extra time.

In most cases of large programs, the compilation time for these analyses, in order to prepare the executable to be ready for incrementality use, needed more than 4 minutes.

Every execution time of a program is different due to the different facts produced.

Below we present the times of the Soufflé versus DDlog engine, which were using self-contained analysis, in an increasing order of CPU's numbers. For much faster and instantly result, Soufflé is chosen to be executed in compilation mode instead of interpretation mode and DDlog to be executed with CLI than its provided API.

Generally, every execution time of a program is different due to the different facts produced. These programs's implementation concerns the non-incremental case. Initially we present the times and a graphic representation of them before the improvement of string interning and meta-attribute is made and then, the times when it has already been implemented. We observe that prior the improvement, DDlog's times were 5-6 times greater than Soufflé. While the improvement is made, it reduced DDlog's times to be just 3 times larger than the Soufflé.

An illustrative example used is the Weka program. Weka [25] (Waikato Environment for Knowledge Analysis) is a suite of machine learning software written in Java, Weka contains a group of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces for easy access to these functions.

Jobs/Workers	*Ddlog(CLI mode)*	*Souffle(compilation)*
1	8m 27, 263s	1m 10, 796s
2	7m 1, 683s	1m 0, 736s
4	5m 53, 323s	0m 50, 897s
6	5m 22, 058s	0m 45, 584s
8	5m 4, 872s	0m 43, 141s
10	5m 6, 297s	0m 42, 861s
12	4m 57, 898s	0m 41, 094s
14	5m 0, 732s	0m 41, 838s
16	4m 57, 453s	0m 44, 144s
20	5m 18, 739s	0m 41, 068s
24	5m 33,025s	0m 39, 245s

Figure 1: Time comparison (DDlog vs Soufflé) before improvement

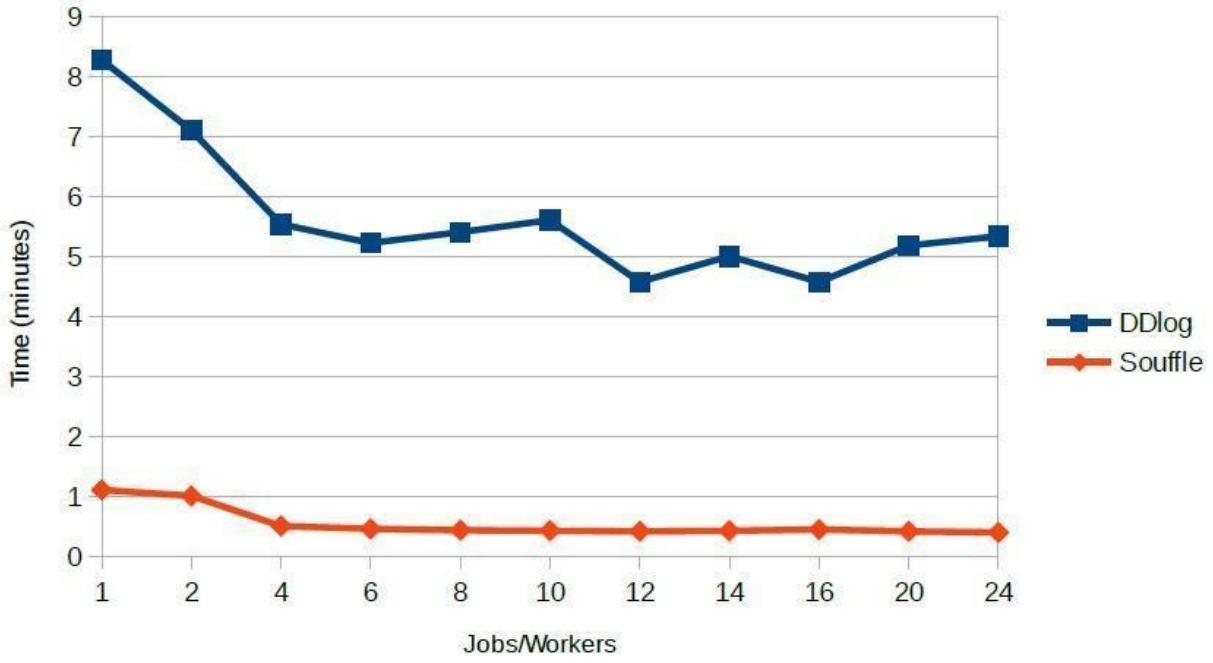

Figure 2: Graphical representation of comparison time (Ddlog vs Soufflé)

Weka:

Jobs/Workers	DDlog(CLI)	Souffle(Compile)
1	3 m 13, 570 s	0 m 57, 576 s
2	2 m 28, 499 s	0 m 49, 535 s
4	2 m 24, 458 s	0 m 43, 340 s
6	2 m 19, 464 s	0 m 37, 853 s
8	2 m 19, 741 s	0 m 39, 894 s
10	2 m 16, 515 s	0 m 39, 481 s
12	2 m 13, 970 s	0 m 41, 393 s
14	2 m 16, 890 s	0 m 40, 471 s
16	2 m 13, 046 s	0 m 40, 436 s
20	2 m 19, 195 s	0 m 40, 538 s
24	2 m 23, 155 s	0 m 41, 490 s

Figure 3: Time comparison (DDLog vs Soufflé) after the improvement

Afterwards, we present the times, in non-incremental case, it took the Weka program to compute the micro analysis with the DDlog's improvements. In this case, the time difference between DDlog and Soufflé has grown; DDlog was 4 times greater than Soufflé, meaning that micro analysis logic is greater than the self-contained, and one reason is because micro includes self-contained.

Weka:

Jobs/Workers	DDlog(CLI)	Souffle(Compile)
1	4 m 42, 639 s	1 m 49, 017 s
2	3 m 55, 134 s	1 m 18, 291 s
4	3 m 42, 296 s	1 m 0, 492 s
6	3 m 40, 174 s	0 m 53, 659 s
8	3 m 39, 550 s	0 m 50, 510 s
10	3 m 42, 081 s	0 m 50, 122 s
12	3 m 33, 709 s	0 m 52, 302 s
14	3 m 35, 701 s	0 m 49, 213 s
16	3 m 43, 278 s	0 m 51, 223 s
20	4 m 1, 119 s	0 m 47, 697 s
24	4 m 4, 256 s	0 m 47, 469 s

Figure 4: Micro's analysis time comparison (DDlog vs Soufflé)

In addition, we present a graphical representation of indicative memory consumption, in non-incremental case, of Weka program using self-contained analysis in 4 and 16 CPUs. Due to the property of incrementality the memory that was bound by DDlog is much larger than the Soufflé, as shown below.

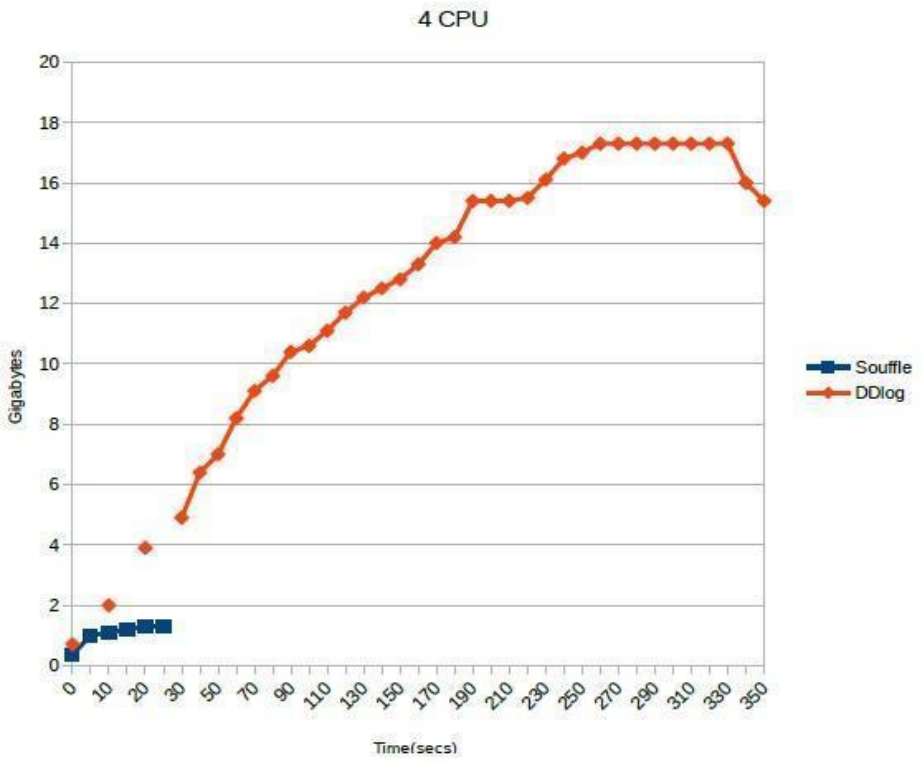

Figure 5: Memory consumption (4 CPUs)

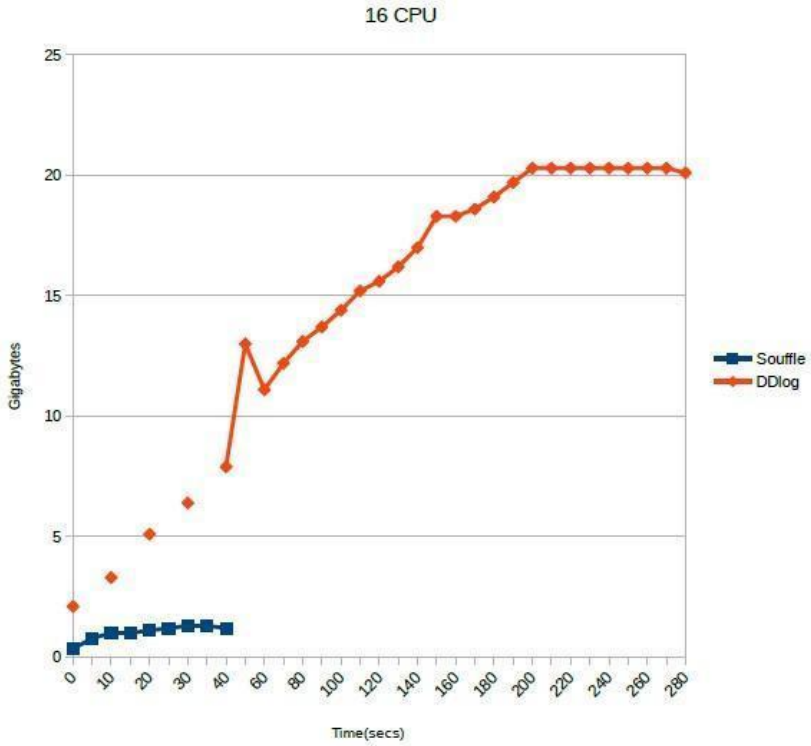

Figure 6: Memory Consumption (16 CPUs)

To show incrementality cases of micro analysis, we were deleting or inserting (one at a time, ten at a time, a hundred at a time and a thousand at a time) facts of methods invocations (Special, Static and Virtual), to see how DDlog will react in these changes.

One program we used is Ant for this incremental case by deleting or inserting one at a time facts of the Special Method Invocation. We present a graphical representation of deletions or insertions and the decreasing order of them. Ant [26] is a software tool for automating software build processes, which implemented using the Java language, requires the Java platform, and is best suited to building Java projects.

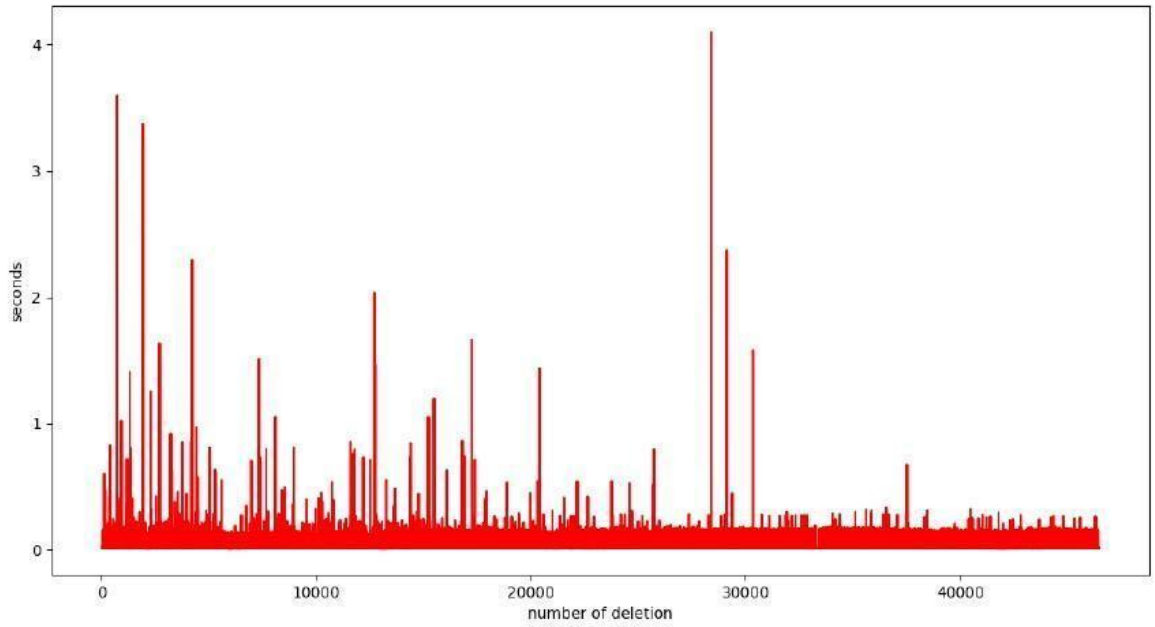

Figure 7: Graphical representation of deletions of Ant's Special Method Invocation

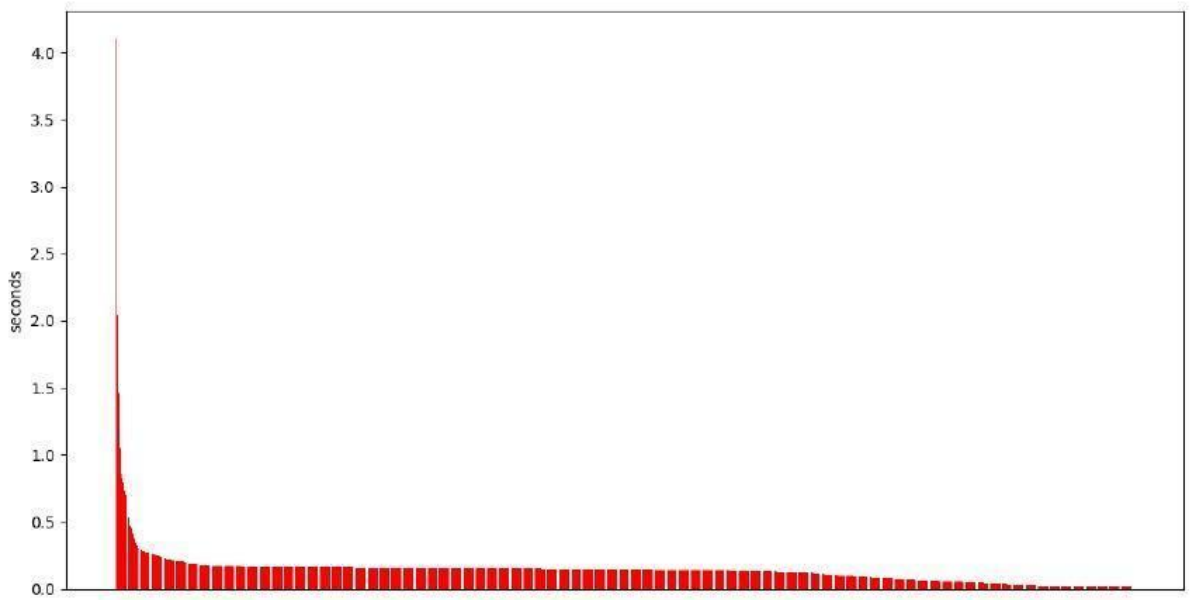

Figure 8: Graphical representation of deletions in descending order (10% of the sample)

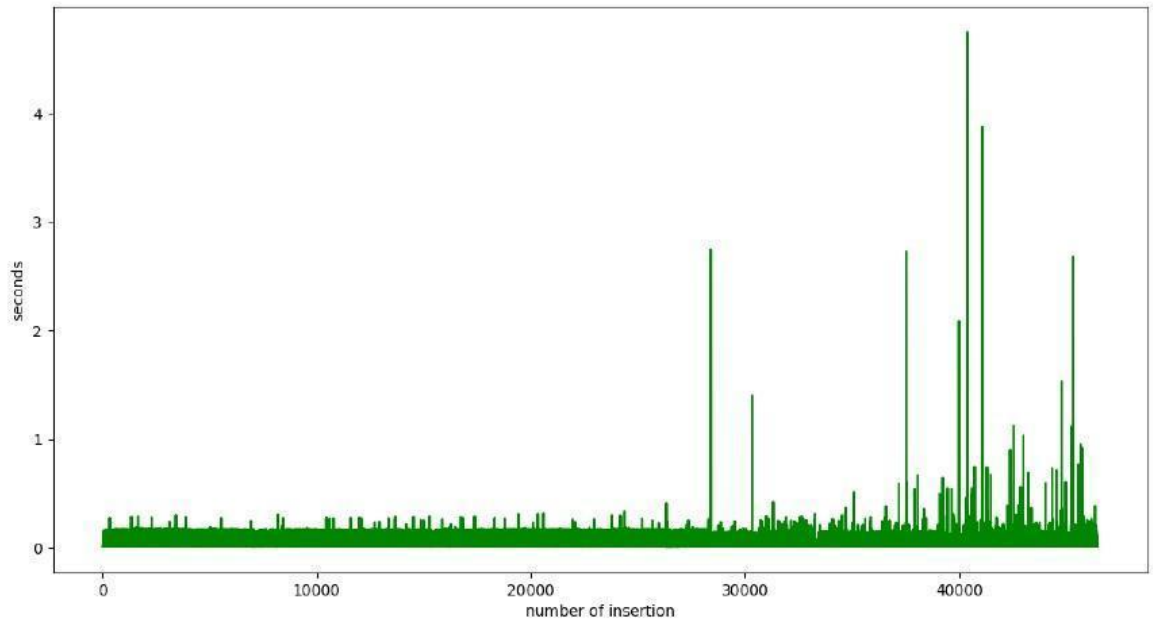

Figure 9: Representation of insertions of Ant's Special Method Invocation

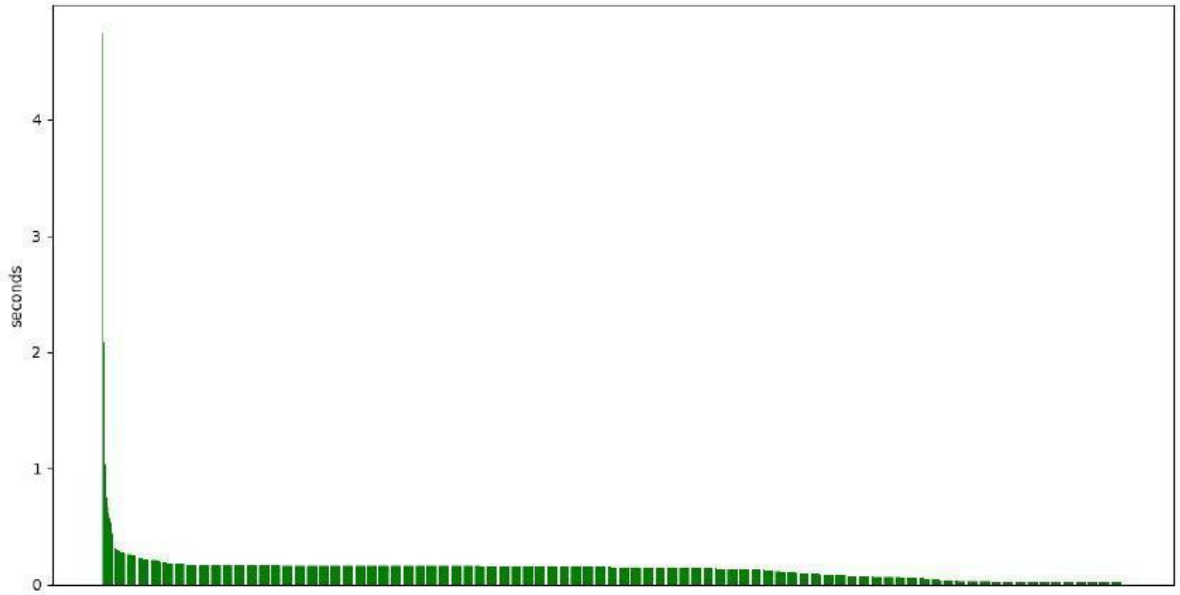

Figure 10: Graphical representation of insertions in descending order (10% of the sample)

All the above times were without the extra time of dumping the results. Initial time of execution of program Ant without the incremental parts is 0 m 48, 360s. The size of Special Method Invocation facts was 46475 lines. For the above case the total time of execution was 37 m 47, 490s, where 18 m 50s was for deletion time and 18 m 7s for insertion time.

Another program we used is Batik for this incremental case by deleting or inserting thousand at a time facts of the Virtual Method Invocation and we present a graphical representation of these deletions or insertions and the decreasing order of them. Batik [27] is a pure-Java library that can be used to render, generate, and manipulate SVG graphics. A Scalable Vector Graphics (SVG) toolkit that renders a number of SVG files. It uses additional threads to speed the rendering or transcending process.

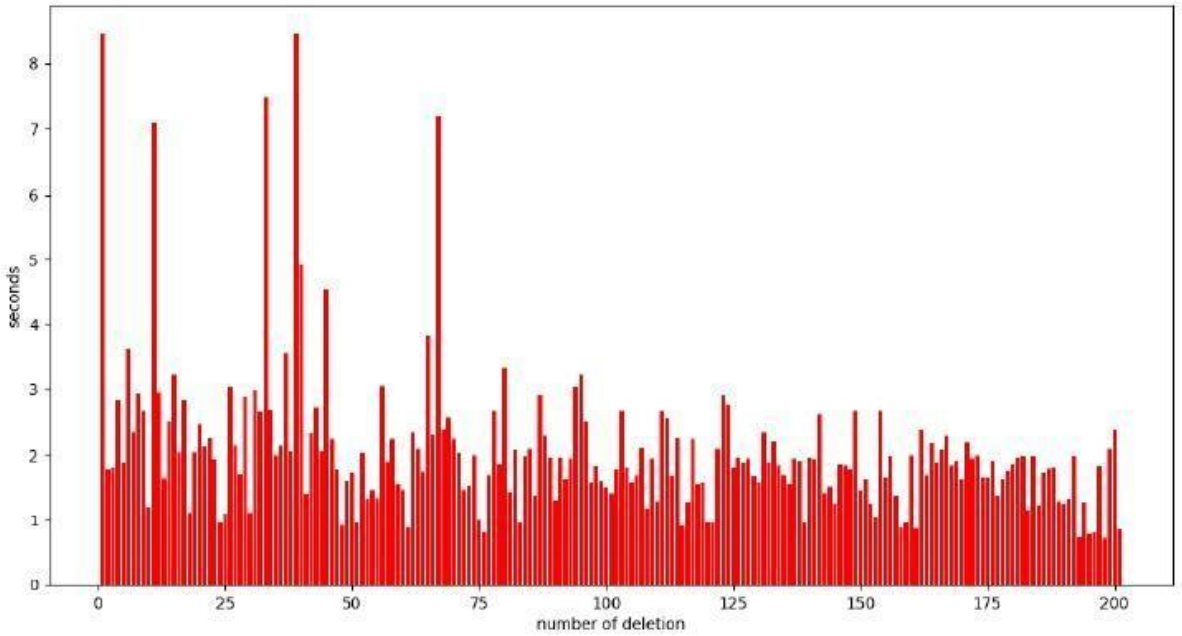

Figure 11: Graphical representation of deletions of Batik's Virtual Method Invocation

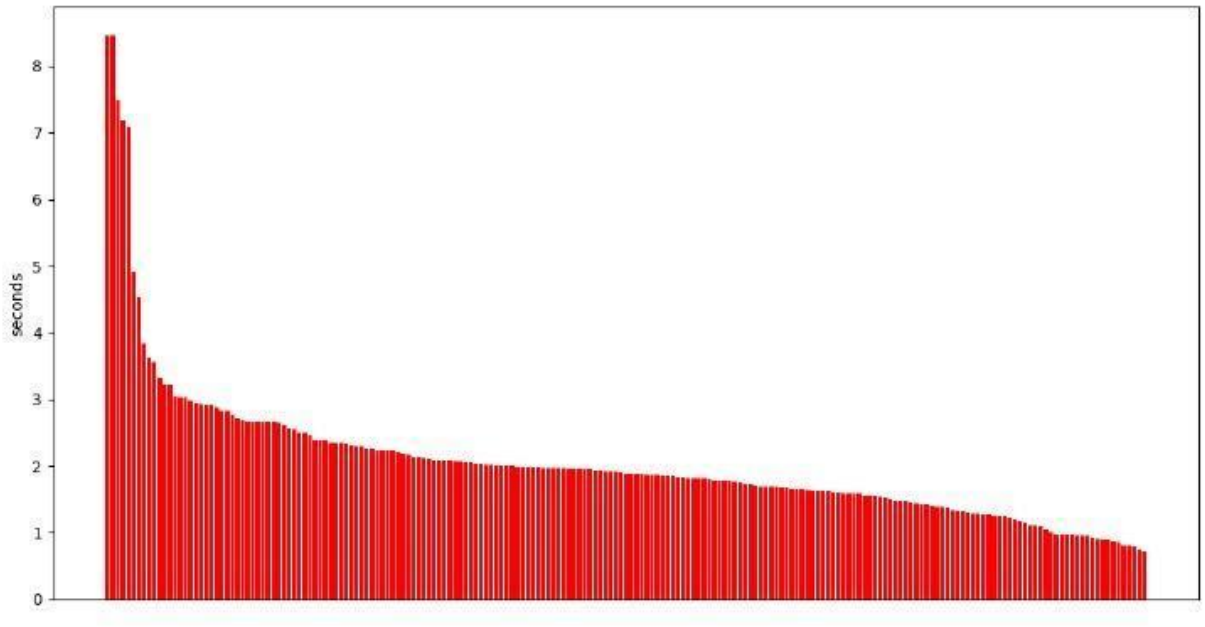

Figure 12: Graphical representation of deletions in descending order

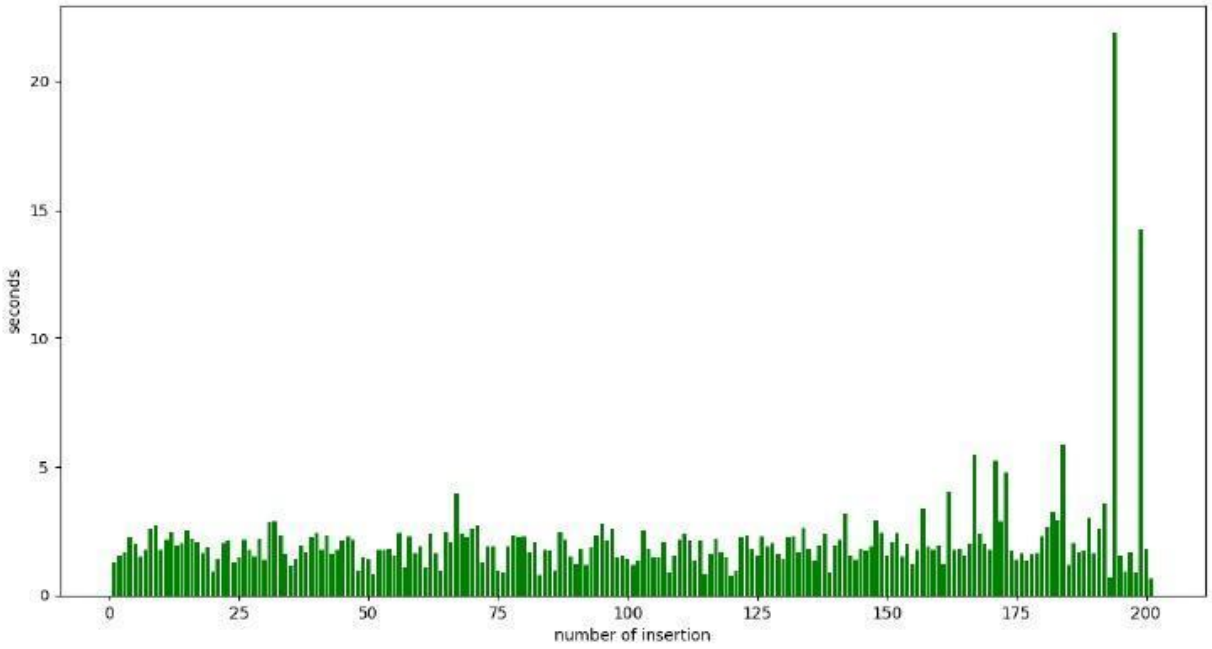

Figure 13: Graphical representation of insertions of Batik's Virtual Method Invocation

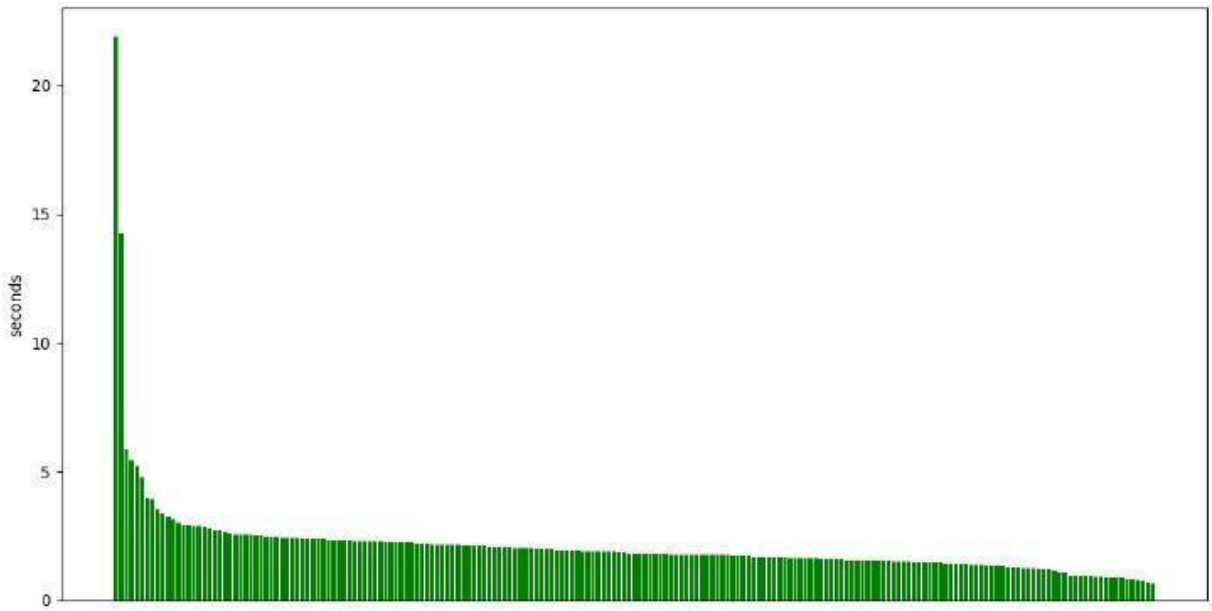

Figure 14: Graphical representation of insertions in descending order

All the above times were without the extra time of dumping the output results. The initial time of Batik's execution without the incremental parts is 1 m 46, 39s. The size of Virtual Method Invocation facts was 200143 lines. For the above case the total time of execution was 15 m 47, 490s, where 6 m 54s was for deletion time and 7 m 2s for insertion time.

5. CONCLUSION

DDlog offers an alternative incremental computation for static analysis. With DDlog and Doop we could test incremental aspects of static analyses and see how effective they could be.

Although our benchmarks yield some positive results, further work is necessary to achieve the desirable times, to match Soufflé's optimal ones. Differential Datalog depends on incremental algorithms that maintain intermediate results of computation. Incremental Datalog implementations may not yield practical benefit in most realistic scenarios, in the presence of complex recursion. Sometimes the worst case complexity of an incremental algorithm is the same as that of full computation. But, this is not to say that there aren't scenarios where incrementality will be effective.

DDlog minimizes the needed conversions between various languages in large projects and focuses on continuous improvement of its performance and memory utilization, so as to be able to manage successfully every demanding incremental case.

We hope that this work could lay the groundwork for future research on incremental execution of static analysis. Developers should adopt programming tools such as DDlog, which solve the incremental computation's complexity once and for all, rather than addressing it on case-by-case basis.

ABBREVIATIONS - ACRONYMS

API	Application Programming Interface
EDB	Extensional Database
IDB	Intensional Database
DDlog	Differential Datalog
DD	Differential Dataflow
CPU	Central Processing Unit
RAM	Random-Access Memory
OS	Operating System

REFERENCES

[1] Y. Smaragdakis and G. Balatsouras, “Pointer Analysis”, Foundations and Trends R in Programming Languages, vol. 2, no. 1, pp. 1–69, 2015. Available: http://yanniss.github.io/points-to-tutorial15.pdf

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pages 243–262, 2009. Available: https://yanniss.github.io/doop-oopsla09prelim.pdf

[3] L. Ryzhyk and M. Budiu. Differential Datalog (DDLog). VMware Research, https://github.com/vmware/differential-datalog [Online]. Retrieved June 2019.

[4] “Home · oracle/souffle Wiki · GitHub” [Online] Available: https://github.com/oracle/souffle/wiki

[5] F. McSherry. Differential Dataflow. https://github.com/TimelyDataflow/differential-dataflow . Retrieved January 2019.

[6] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In Conference on Innovative Data Systems Research (CIDR), Microsoft Research, Silicon Valley Lab, January 2013.

[7] B. Scholz, H. Jordan, P. Subotic, and A. Jordan, “Engineering Static Analyzers with Soufflé”, pp. 17-93. Retrieved June 2019. Available: https://souffle-lang.github.io/pdf/SouffllePLDITutorial.pdf

[8] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, “Static Analysis of Java Dynamic Proxies”, ISSTA’18, July 16–21, 2018, Amsterdam, Netherlands. Retrieved June 2019. Available: https://yanniss.github.io/issta18-dynamic-proxies-preprint.pdf

[9] F. McSherry, “Differential Datalog”, https://github.com/frankmcsherry/blog/blob/master/posts/2016-06-21.md. Retrieved June 2019.

[10] F. McSherry, “An introduction to differential dataflow, part 1”, https://github.com/frankmcsherry/blog/blob/master/posts/2015-09-29.md. Retrieved June 2019.

[11] F. McSherry, “An introduction to differential dataflow, part 2”, https://github.com/frankmcsherry/blog/blob/master/posts/2015-11-27.md. Retrieved June 2019.

[12] Y. Smaragdakis Lecture, Topic: “Declarative Static Program Analysis”, Department of Informatics-AUEB, Athens, Greece, May 29, 2013. Available: http://cslab252.cs.aueb.gr/el/content/declarative-static-program-analysis. Retrieved June 2019.

[13] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, “Porting Doop to Souffle:´ A Tale of Inter-Engine Portability for Datalog-Based Analyses”, Department of Informatics University of Athens, SOAP’17, June 18, 2017, Barcelona, Spain. Available: https://yanniss.github.io/doop2souffle-soap17.pdf. Retrieved June 2019.

[14] T. J. Green, S. Huang, B. T. Loo, W. Zhou. “Datalog and Recursive Query Processing”. Foundations and Trends in Databases, vol. 5, no. 2, pp. 105–195, 2012. Available: http://blogs.evergreen.edu/sosw/files/2014/04/Green-Vol5-DBS-017.pdf. Retrieved June 2019.

[15] ”Datalog” [Online] Available: https://en.wikipedia.org/wiki/Datalog. Retrieved June 2019.

[16] "Crate differential_dataflow - Rust" Available: https://docs.rs/differential-dataflow/0.9.0/differential_dataflow/. Retrieved June 2019.

[17] F. McSherry, "Differential dataflow", Apr 7, 2015, Available: http://www.frankmcsherry.org/differential/dataflow/2015/04/07/differential.html. Retrieved June 2019.

[18] A. Colyer, "Differential Dataflow," in the morning paper, [online], JUNE 17, 2015. Available: Reference Online, https://blog.acolyer.org/2015/06/17/differential-dataflow/. Retrieved June, 2019.

[19] M. Abadi, F. McSherry, and G. D. Plotkin, "Foundations of Differential Dataflow", Microsoft Research, LFCS, School of Informatics, University of Edinburgh, Available: http://homepages.inf.ed.ac.uk/gdp/publications/differentialweb.pdf

[20] L. Ryzhyk and M. Budiu. A differential Datalog (DDLog) tutorial. https://github.com/ryzhyk/differential-datalog/blob/master/doc/ tutorial/tutorial.md. Retrieved January 2019.

[21] L. Ryzhyk and M. Budiu. Differential Datalog (DDLog) language reference. https://github.com/ryzhyk/differential-datalog/blob/master/doc/language_reference/language_reference.md. Retrieved January 2019.

[22] L. Ryzhyk and M. Budiu. Creating Datalog Tests https://github.com/vmware/differential-datalog/blob/master/doc/testing/testing.md. Retrieved January 2019.

[23] Leonid Ryzhyk and Mihai Budiu, "Differential Datalog", VMware Research, Available: https://github.com/vmware/differential-datalog/blob/master/doc/datalog2.0-workshop/paper.pdf. Retrieved January 2019.

[24] "Incremental Computing" [Online] Available: https://en.wikipedia.org/wiki/Incremental_computing. Retrieved June 2019.

[25] "Weka (machine learning)" [Online] Available: https://en.wikipedia.org/wiki/Weka_(machine_learning). Retrieved June 2019.

[26] "Apache Ant" [Online] Available: https://en.wikipedia.org/wiki/Apache_Ant. Retrieved June 2019.

[27] "Apache Batik" [Online] Available: https://en.wikipedia.org/wiki/Apache_Batik, Retrieved June 2019.