NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSGRADUATE STUDIES

MASTER THESIS

Performance optimization on Declarative Points-to Analysis using the Souffle Datalog Engine

Christos N. Zisis

Supervisors: **Yannis Smaragdakis,** Professor NKUA
Anastasios Antoniadis, PhD. Candidate NKUA

ATHENS

MARCH 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΉΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση απόδοσης Δηλωτικής Ανάλυσης Δεικτών με τη χρήση της Datalog διαλέκτου του Souffle

Χρήστος Ν. Ζήσης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Αναστάσιος Αντωνιάδης, Υποψήφιος Διδάκτορας ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2019

MASTER THESIS

Performance optimization on Declarative Points-to Analysis using the Souffle Datalog Engine

Christos N. Zisis
A.M.: M1593

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
Anastasios Antoniadis, PhD. Candidate NKUA

EXAMINATION COMITEE: **Yannis Smaragdakis,** Professor NKUA
Alex Delis, Professor NKUA

ATHENS

MARCH 2019

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση απόδοσης Δηλωτικής Ανάλυσης Δεικτών με τη χρήση της Datalog διαλέκτου του Souffle

Χρήστος Ν. Ζήσης
Α.Μ.: Μ1593

ΕΠΙΒΛΕΠΟΝΤΕΣ **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Αναστάσιος Αντωνιάδης, Υποψήφιος Διδάκτορας ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Αλέξης Δελής, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2019

ABSTRACT

CClyzer is a tool that aims to provide a complete points-to analysis on LLVM bitcode while, at the same time, providing context-sensitivity variance. While it was originally implemented by making use of the LogicBlox Datalog engine and the LogiQL dialect, in this documentation we present our work of porting CClyzer to a new Datalog engine- Souffle. Studying the analysis CClyzer provides on C/C++ LLVM bitcode, we believe Souffle is a tool that enhances CClyzer's logic in order to minimize analysis running times. Our evaluation shows promising results as the analysis running time is minimized by 95% on small programs while parallel execution adds an average speedup of 1.29 for bigger workloads.

SUBJECT AREA: Static program analysis

KEYWORDS: static program analysis, cclyzer, souffle, structure-sensitivity, context-sensitivity, llvm

ΠΕΡΙΛΗΨΗ

Το CClyzer είναι ένα εργαλείο το οποίο έχει σκοπό να παρέχει μια ολοκληρωμένη ανάλυση δεικτών σε κώδικα LLVM και σε συνάρτηση με μεταβαλλόμενη ακρίβεια ως προς την ιεραρχία κλήσεων κάθε μεθόδου. Ενώ αρχικά υλοποιήθηκε με χρήση της LogicBlox μηχανής για τη γλώσσα Datalog και τη διάλεκτο LogiQL, σε αυτή την εργασία παρουσιάζουμε την μεταφορά του CClyzer σε ένα νέο περιβάλλον για εκτέλεση προγραμμάτων σε Datalog, το Souffle. Μελετώντας την ανάλυση την οποία προσφέρει το CClyzer σε LLVM bitcode που προκύπτει από πηγαίο κώδικα γλώσσας C και C++, πιστεύουμε ότι το Souffle αποτελεί πολύτιμο εργαλείο στην ενίσχυση της λογικής λειτουργίας του CClyzer με τελικό στόχο την ελαχιστοποίηση του χρόνου εκτέλεσης της ανάλυσης. Οι μετρήσεις μας δείχνουν ενθαρρυντικά αποτελέσματα καθώς ο συνολικός χρόνος εκτέλεσης της ανάλυσης μειώνεται κατά 95% για μικρά προγράμματα ενώ η εκτέλεση με τη χρήση παραλληλίας προσφέρει, κατά μέσο όρο, 29% επιτάχυνση για μεγαλύτερα.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική ανάλυση προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση προγραμμάτων, cclyzer, souffle, δομική ευαισθησία, ευαισθησία συμφραζόμενων, llvm

AKNOWLEDGEMENTS

In this section, I would like to thank my advisor, Professor Yannis Smaragdakis for the guidance and all the insightful comments he provided and also the patience and trust he displayed in me for the completion of this work. It is my honor to have collaborated with him.

I would also like to thank my father, mother and my siblings: Melina, Nondas and Alex for all of their continuous love and support. My friends Nikos, Marilia, Anastasis, Vaggelis, Giannis, Manos, Marilena and Martin from whom I have received nothing but constant encouragement and inspiration to move forward.

Finally, I would like to thank every member of the PLAST lab who were always willing to share their knowledge and expertise.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This documentation constitutes our search on how static analysis programs, written in Datalog, can be optimized using engines that take advantage of parallelism. More specifically we examine how our static analysis tool behaves on two different engines while presenting their respective differences. It was developed as a Master Thesis on the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, under the supervision of Professor Yannis Smaragdakis and PhD Candidate Tony Antoniadis.

1. INTRODUCTION

Datalog is a declarative language used to define relations between various entities we use in our analysis. It is considered an ideal option to describe the rules in a program analysis because of its expressiveness and functionality. In Datalog we define rules in the form of:

That(...) ← This(...) .

This is a rule that may be used to describe the expression “If This(...), then That(...)”. This(...) is considered to be the body of the rule, a relation which the engine checks for existence in our knowledge base. If in fact This(...) is true then Datalog will also add the rule’s head—That(...)—in our knowledge base. In Datalog we use entities as arguments in the rules we declare. For example: That(x) :- This(x). Here we conclude that if This(x) is true then That(x) should also be true, “x” being a variable/entity in our Datalog program. We may also define rules such as: A(x) :- B(x), C(x). Which in turn describes that for A(x) to be true, we need both B and C to be true for the same variable “x”. Obviously, we can also describe the union of two rules with the following example:

A(x) ←B(x). **(1)** A(x) ← B(x) ; C(x). **(2)**
A(x) ←C(x).

Here (1) and (2) describe the expression “A is true for x, if either B or C is true for x”. It is worth mentioning that while (1) is the standard way of describing a union in Datalog, (2) is also acceptable by Souffle as a syntactic feature and we may use it to describe small, non-complex rules in our analysis. We may declare many rules such as the previous examples. These will, in turn, be evaluated multiple times with each one expanding our knowledge base. When a fix-point is reached the evaluation will terminate and we may examine our final results.

The rest of the thesis is organized as follows:

- In Chapter 2 we present some background on CClyzer, Souffle and the LogicBlox Datalog Engine
- In Chapter 3 we introduce the differences, both in technical and dialect-level, between our comparing implementations on each of the engines; Souffle and LogicBlox.
- In Chapter 4 we perform all of our evaluations and comparisons for CClyzer both in Souffle and LogicBlox and discuss the design details that support them.
- In Chapter 5 we summarize our conclusions.

2. CCLYZER, SOUFFLE AND THE LOGICBLOX DATALOG ENGINE

2.1 CClyzer

CClyzer [1] is a tool providing static analyses written in Datalog, applying over LLVM IR (an intermediate language for a popular compiler family) produced by C/C++ programs. Its initial implementation uses the LogicBlox Datalog Engine, adopting the LogiQL language and was written in three variants to implement multiple levels of context sensitivity. Currently it supports:

- context insensitivity,
- 1-call-site sensitivity with a heap context,
- 2-call-site sensitivity with a heap context.

Our implementation aims to support all the three aforementioned variants to provide the same precision levels as the original implementation in LogiQL; the dialect LogicBlox uses. At the same time we aim to achieve much higher performance in terms of execution time, taking advantage of parallelism and to provide enough context abstractions for one analysis to support interchangable context sensitivity.

Context sensitivity aside, CClyzer provides a structure sensitive points-to [2] analysis for C/C++ LLVM bitcode that recovers much of the available high-level structure information of types and objects, by applying two key techniques:
(1) It records the type of each abstract object whenever this is available. Such cases include the usual appearance of stack allocations, global variables and calls to default or user-defined constructors through the call of the new() method in C++. In cases when the type is not readily available (as is the case with malloc() method in C), the analysis creates multiple abstract objects for each type the allocated object could have.
(2) It creates separate abstract objects that represent:

a) The fields of objects of either struct or class type. In this sense, given an abstract object O with type T, the analysis will create objects O.fi for each i-th field defined in type T.

b) The (statically present) constant indices of arrays, resulting in a limited form of array-sensitivity. Given an abstract array object, called A with size S, the analysis will create objects A[i] for each i with value 0 to S-1 and a new object called A[*] to account for all unknown indices met. These usually include variable values or expressions in the array's index, that cannot be determined before runtime.

2.2 The two engines

LogicBlox as well as Souffle programs, act as databases that contain both a relational schema and its data. In particular, databases hold collections of facts, each of which is concerned with a predicate. In logic, predicates are either properties that may be held by individual variables or relationships that may apply between multiple entities. In a database, a collection of facts associated with a predicate is called that predicate's population. Sometimes the distinction between the logical predicate and its population is glossed over by referring to the stored predicate population simply as a predicate. In the following sections we discuss differences between the two, to better understand each functionality and dialect.

2.3 LogicBlox Datalog Engine

The LogicBlox Datalog Engine acts as a database manager that evaluates and expands upon its knowledge base. The initial facts constitute the extensional database (EDB). After they are inserted in the knowledge base, the logic defined within the program's Datalog rules is executed until a fix-point is reached. The knowledge base is expanded with the new facts produced composing our intensional database (IDB). These in turn, create new relations between already imported entities or newly created abstract objects. A subset of those relations, which the program defines as output, is exported as long as its entities are in human-readable form.

2.4 The Souffle Engine

Souffle [3] is short for "Systematic, Ontological, Undiscovered Fact Finding Logic Engine" and is a variant of Datalog for tool designers crafting static analyses. Each program is transformed to a relational algebra machine applying a Futamura Projection on the semi-naive evaluation scheme and the IDB. The relational algebra is then compiled into a highly templatized C++ parallel optimized source code and in turn, executable. The compiled program then reads facts from a disk-based format, executes the IDB logic and outputs the results in the same disk-based format as the EDB.

3. TECHNICAL IMPLEMENTATION DIFFERENCES

3.1 Execution

The semantics of the LogicBlox Datalog system [4] allow it to interleave declarative evaluation with imperative execution. The LogicBlox engine was created to support delta logic; the addition and removal of facts from the database's population. Before the transaction performing such an update commits, each rule's logic and population is evaluated incrementally, in order to maintain consistency in the database. This delta logic is used in order to edit facts in our analyses and thus the engine favors serialized execution in order to support it.
In contrast, Souffle operates as a batch processor of tuples, in a disk-based format. Predicates are read in, computation is performed, and the program determines which "output" predicates will be exported. However, Souffle does not provide a way to retract tuples from a relation during execution, nor a way to keep a state of computation, incrementally change inputs, and re-evaluate. It mainly focuses on shared-memory parallelism to take advantage of multithreading, in order to minimize run-time. However, the LogicBlox implementation of CClyzer uses delta logic only to import initial facts. In fact, it can easily be replaced with Souffle's default import mechanism without the constraint of serial execution.

3.2 Rules and constraints

A rule to fill a predicate with facts, in LogiQL, may be written as:

$A(x) \leftarrow B(x).$

Here the ← symbol separates the established facts of predicate B from the not-yet evaluated facts of predicate A. However another symbol exists. We may write down a rule such as:

$A(x) \rightarrow B(x).$

We mainly use the → symbol to express constraints in the sense that "If A(x) is true, it is impossible for B(x) to not be true". Here → maps each entity (i.e., variable x) from the body to the head side essentially acting as a function. The engine, then, will detect at run-time any attempt to violate the functional constraint.
On the other hand, Souffle deals with these symbols in a different way. To write a declarative rule between two predicates in Souffle we have to write:

A(x) :- B(x).

Here :- is the symbol that define rules between relations and there is no inherent mechanism to support constraints. However, we may simulate the constraint mechanism using a different mechanism provided by Souffle. Since we use constraints mainly for debugging, we define the predicate schema_sanity(). We then append it to the body of any rule that participates in the analysis for the sole purpose of testing. While schema_sanity() is inserted in our knowledge base, these rules will produce any

output that violates our constraints. To understand its use, we give the following example:

A(x) :- schema_sanity(), !B(x).

In this case the engine will append any x for which B(x) is false into the population of predicate A. If such a result is found it will be exported as a warning that the restriction set was not met during execution to inform the developer of logical errors. Of course, in this case we have to define any possible violation instead of the constraint.

3.3 Type system

LogiQL is a strongly-typed language with dynamic tagging of types. While every value has a unique “principal” type, this type is unknown to the programmer, since the engine is responsible for the creation of entities. For each entity, we do not have any information about its type or contents. The only exception to this rule are our initially inserted variables in predicates that act as reference to each entity participating in the evaluation of the IDB logic. Each of these values can be queried (e.g., in a rule body) to retrieve its type (which is unique, per strong typing). For example:

A(x) → .

While the above declaration, defines a constraint, x is not bounded by anything. There is no information given about how A entities are represented in-memory. The LogiQL engine will handle the internal representation automatically. However, while inserting our initial facts we know the values populating each predicate.

A(x), referenceA(x:s) -> string(s).

Here, +A(“foo”) will create a new entity for A, other than “foo”. The predicate referenceA acts as refmode; a reference matching the newly created entity x with “foo”, while decrlaring a constraint that binds s as a string.
Moreover, LogicBlox performs type inference so that most predicates do not need to have their types of their variables declared, so long as the types of the variables can be inferred by the predicates involved in the rules that populate them.
On the other hand, Souffle distinguishes symbols and numbers, but otherwise allows constants to be used as values of any compatible type. The program’s execution carries no dynamic type information and all predicates need to have full declarations.
In our next example we define the same rule in LogiQL and Souffle respectively:

LogiQL:
Context(?context) → .
FunctionDecl(?function) → .
ReachableContext(?context, ?function) → Context(?context), FunctionDecl(?function).

Souffle:
.type FunctionDecl
.type Context
.decl ReachableContext(?context: Context, ?function: FunctionDecl)

Note that, in LogiQL, by making use of the type inference mechanism the engine provides we do not have to explicitly define the type of any argument. The engine defines it by recursively infering the definitions of each predicate participating in the right side of the constraint.
In Souffle we have to define our types using **.type** and declare each type used in our predicate ReachableContext. We then use **.decl** to define each of the types our predicate is allowed to use. In this aspect, Souffle is able to warn us at compile-time about any abnormalities observed in our program, regarding the types used. Moreover, we have a type system that supports sub-typing, with our basic types being symbol (character string) and number. We may introduce type hierarchies such as:

```
.type Instruction = symbol
.type CallInstruction = Instruction
```

describing that type Call-Instruction is a sub-type of Instruction which, in turn, is a sub-type of symbol. Both hold the same basic information: Call-Instruction and Instruction are character strings. Finally, this makes room for a fully functional type-checking system at compilation time.

3.4 Functional Predicates

LogiQL offers the use of a second kind of predicate called “functional predicates”. While their syntax remains the same these are distinguished in their definition, as they use square brackets instead of parentheses. The square brackets indicate the functional nature of the relationship and its variables. We define the following functional predicate:

```
studentOf[name] = s → Student(name), School(s).
```

The use of the word Of in the name of the predicate emphasizes the connection between a student and that student’s school. In contrast, to the functional approach we may define a predicate such as:

```
isStudentOf(name, s) → Student(name), School(s).
```

Note that although isStudentOf and studentOf appear to serve the same purpose, there is a subtle difference. Implicit in the student domain is the constraint that no student can belong to two schools. Now we populate our functional predicate the same way we would do with a classic one; by defining a logic or adding explicit facts to the relationship. However LogiQL offers a third way to populate functional predicates.

3.5 Constructors

Normally, calling studentOf in a rule’s body, with an empty school domain would result in an empty head. But LogiQL offers a default mechanism for the construction of new entities. To utilize this we have to define our functional predicate as a constructor. We do this by writing:

```
lang:constructor(`studentOf).
studentOf[name] = s → Student(name).
```

By transfering our constructor from body to head, we invoke its construction mechanism by which it produces a new school s.

studentOf[name] = s ← Student(name).

In this example a new and unique School s is produced for every student. Again, since our variable s does not participate in the constraint, the engine will will handle the internal representation automatically .In this sense, each of the students in our database belongs to a different school unless we have any information to prove otherwise. Although the default mechanism creates new school entities, these can not serve as output values. They need to be re-matched through a series of rules, with our initially inserted, printable ones. Nevertheless, we are forced to trust LogiQL's default construction mechanism of new entities to match our already established variables.
On the other hand, Souffle supports only tuple constructors. We may create new entities by managing character strings or number variables in order to create either tuples or variables of the same type. In case we want to introduce a new entity at the head of a rule we are forced to also include it's creation logic in the body of the same rule. For example:

```
studentOf(?name,?school):-
      student(?name),
      ?school = concatenate("schoolOf", ?name).
```

In this example we achieve the same functionality as our equivalent LogiQL rule above with the main difference that we now control the creation logic of our new ?school entity. Here we know that ?school is a string which reads "schoolOf" and the name of the student. We could also use tuples to create our entities the following way:

```
studentOf(?name, ?school):-
      student(?name),
      ?school = [?name].
```

Tuples in Souffle are exported after passing through an indexing system. Their final value cannot be read, as they appear as integers in each output file, but otherwise still hold all relational information.

What's more, we may use built-in predicates that help us determine basic string (concatenation, sub-string, match regular expression, equality, length etc.) or arithmetic (addition, multiplication, division, comparison etc.) operations as well as conversion of one type to another. There is an argument to be made that this method either gives more workload or control to the programmer, but not only it keeps our entity population at bay, it also discards the overhead of rematching newly created variables with old ones (as is the case with LogicBlox) since these can be exported. Also, we expose our creation logic as many times, the calculations needed to create our new entities, may be simpler than the one a LogiQL constructor uses.

3.6 Context sensitivity implementation

In LogiQL the same analysis is written differently for each context sensitivity. Given a context variant, we create an abstract object for each series of reachable call/invoke instructions to assume the role of a given examined context. We achieve this by utilizing

the functional constructor predicates LogiQL offers. For better understanding we give the following example for a 1-call-site-sensitive analysis:

```
context_new[Instruction]= newContext → call_instruction(Instruction).
lang:constructor(`context_new).
context_new[Instruction] = newContext ←
      call_instruction(Instruction).
```

The above rule creates a new entity newContext for each Instruction in our call_instruction predicate. As we have already seen, our functional constructor predicates create a new value for each parameter inserted as input. The above rule inserts the entire population of call_instruction as arguments for our functional predicate context_new. This in turn creates new entities, in order for them to be considered a subset of the contexts used in our analysis. For example:

```
context_new[Instruction] = Context,
reachable_method(Context, Method) ←
      call_instruction_calls_method(Instruction, Method).
```

Here, our predicate context_new plays the role of a general constructor. It increases its population each time it receives a new entity as an argument. Of course it goes without saying that if called for the same entity, at different parts of the program, it will produce the same value.
In addition, the definition of context_new constructor changes for different context sensitivity variants. For example in the 2-call-site context variant its definition and logic will transform as follows:

```
context_new[Instruction1, Instruction2]= newContext →
        call_instruction(Instruction1),  call_instruction(Instruction2).
lang:constructor(`context_new).
```

```
context_new[Instruction1,Instruction2] = newContext ←
      call_instruction(Instruction1),
      instruction_in_method(Instruction1, Method),
      call_instruction_calls_method(Instruction2, Method).
```

Here, the size of arguments for our constructor changes for different context-sensitivity and so does the type of our newContext variable. This prevents us from writing a universal logic for all context-sensitivity variants.
CClyzer aims to support multiple variants of context-sensitivity modularly. So in Souffle, we implement CClyzer to serve as an analysis that utilizes abstractions in regard to context-sensitivity. This means that the core pointer analysis remains the same regardless of context-sensitivity variant. What changes is the part of the logic that creates a new context, depending on the specified variant. Essentially, our context creation rules are implemented in such a way that they are considered a black box regarding the rest of our analysis. However, to implement a construction mechanism that works as explained, we cannot make use of tuple constructors in a direct way. For example:

```
.type Instruction,
.type Context = [Instruction]
context_new(Instruction, Context),
reachable_method(Context, Method) :-
        call_instruction_calls_method(Instruction, Method),
        Context = [Instruction].
```

This is an example of the context creation logic directly participating in our 1-call-site sensitive analysis. We would have to change this rule for CClyzer's insensitive and 2-call-site sensitive analysis, respectively in the following examples:

```
.type Instruction
.type Context
context_new(Context),
reachable_method(Context, Method) :-
        function_declaration(Method),
        Context = "<<emptyContext>>".
```

———————

```
.type Instruction,
.type Context = [Instruction, Instruction]
context_new(PreviousInstruction, Instruction, Context),
reachable_method(Context, Method) :-
        call_instruction_calls_method(Instruction, Method),
        instruction_in_function(Instruction, CallerFunction),
        call_instruction_calls_method(PreviousInstruction, CallerFunc),
        Context = [PreviousInstruction, Instruction].
```

Note that this not only changes each rule in our analysis significantly, it also requires changes in the type system as well as different declaration for each of our predicates. To overcome this obstacle we follow a different path. We create two different predicates to replace context_new predicate: context_request and context_response. Our thinking is to create a universal declaration for context_request, indifferent to sensitivity in our analysis. For the time being we define this declaration to be:

```
.decl context_request(callerCtx:Context, invoc: ContextItem)
```

Now in each of our rules with a head that contains context_new, we replace context_new with context_request. Note that context_request only participates in the head of each rule that is common among all analyses.
Our rules dedicated only to context-sensitivity are isolated from the rest of the analysis. Among them we define our context_response predicate.

```
.decl context_response(callerCtx:Context, invoc: ContextItem, ctx: Context)
```

Our predicate context_response has the same declaration as context_request with an additional last argument that serves as our newly constructed context. For each different sensitivity in our program, we define rules to populate our context_response by

duplicating the first two arguments of context_request and then declaring a logic to utilize said arguments, in order to initialize our last argument, the one that serves as our newly constructed context. To visualize this process we give a simplified example of how a LogiQL context_new population rule is transferred in Souffle using our context_request – context_response method:

LogiQL:

```
context_new[CallerCtx, EntityA] = newContext,
Target(newContext, Method) ←
    Examine1(EntityA, Method),
    Examine2(EntityA, CallerCtx).
```

Souffle:

```
context_request(callerCtx, EntityA),
Target_Intermediate(callerCtx, EntityA, Method) :-
    Examine1(EntityA, Method),
    Examine2(EntityA, callerCtx).


Target(newContext, Method) :-
    Target_Intermediate(callerCtx, EntityA, Method),
    context_response(callerCtx, EntityA, newContext).
```

Then a context_response rule would be declared, separated from the rest of our analysis as follows:

```
context_response(callerCtx, EntityA, newContext) :-
    context_request(callerCtx, EntityA),
    newContext = (…) callerCtx (…) EntityA.
```

In LogiQL, Examine1 & Examine2 populate Target, after context_new has created our new context. In Souffle, Examine1 & Examine2 populate context_request, along with an intermediate predicate we define as Target_Intermediate. Then context_response creates our new context and finally, Target is filled by the resulting population of Target_Intermediate & context_response.
In our context-related part of the analysis, not only are the rules implemented differently for each context-sensitivity but types also change depending on the quantity of information each variable needs to hold. In an insensitive analysis, for example, a context is just a character string with the value “<<emptyContext>>” while in a 2-call-sensitive analysis a context is declared as a tuple of two separate invocations. Implementation-wise, this affects only our context-related part of the analysis. The rest of our program need only transfer a context-variable between a rule’s body and head without ever touching its contents.

4. EVALUATION

4.1 Comparison

We evaluate each implementation of CClyzer both on Souffle and the LogicBlox Datalog engine with execution time as our metric. Since we use the same fact generator on both implementations, we discard this time from the following comparison.
Souffle provides two different modes of execution. A compiled C++ parallel program and a direct interpreter. The interpreter is the default option when invoking Souffle as a command line tool. When Souffle is invoked in interpreter mode, the parser translates the Datalog program to a RAM program, and executes the RAM program on-the-fly. For computationally intensive Datalog programs, the interpretation is slower than the compilation to C++. However, the interpreter has no costs for compiling a RAM program to C++ and invoking the C++ compiler, which is expensive for larger programs (in the order of minutes).
All experiments are run on an Intel(R) Xeon(R) CPU E5-2667 v2 3.30GHz CPU machine with 256GB of RAM. 16 CPU cores are available, or 32 with hyperthreading. We also run our experiments for all variations of context-sensitivity, in order to observe how it affects runtime.
Below we show a direct comparison between Souffle's Interpreter, Souffle (single-thread) and LogiQL run on the bitcode of 100 Unix core utilities, ordered by Souffle times .

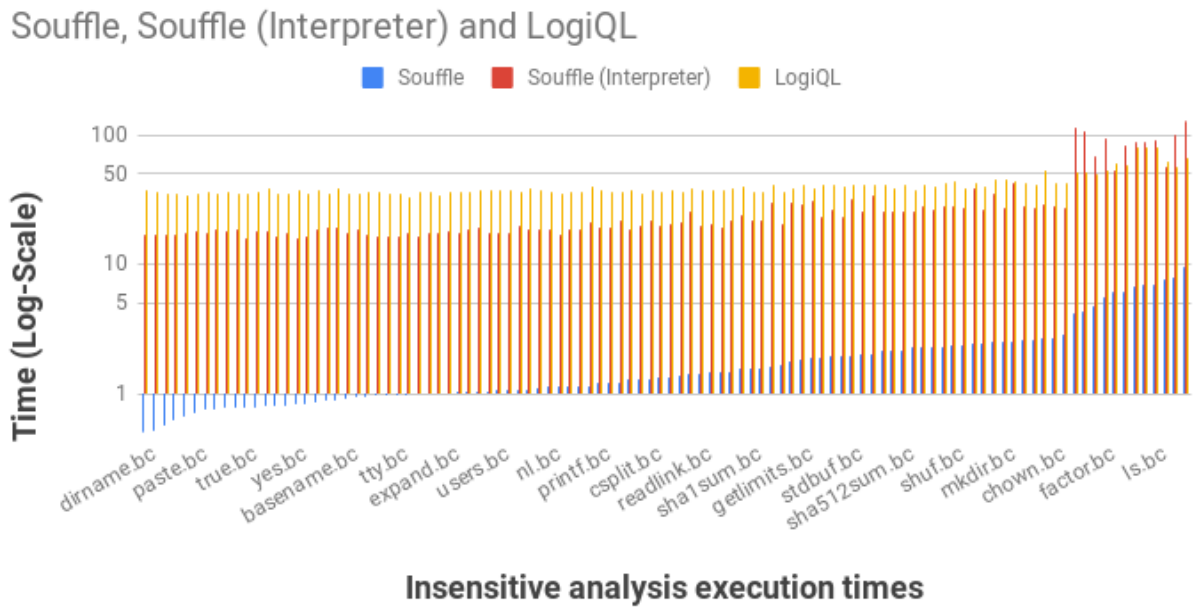

Figure 1: Context-Insenstive Analysis running times (sec.)

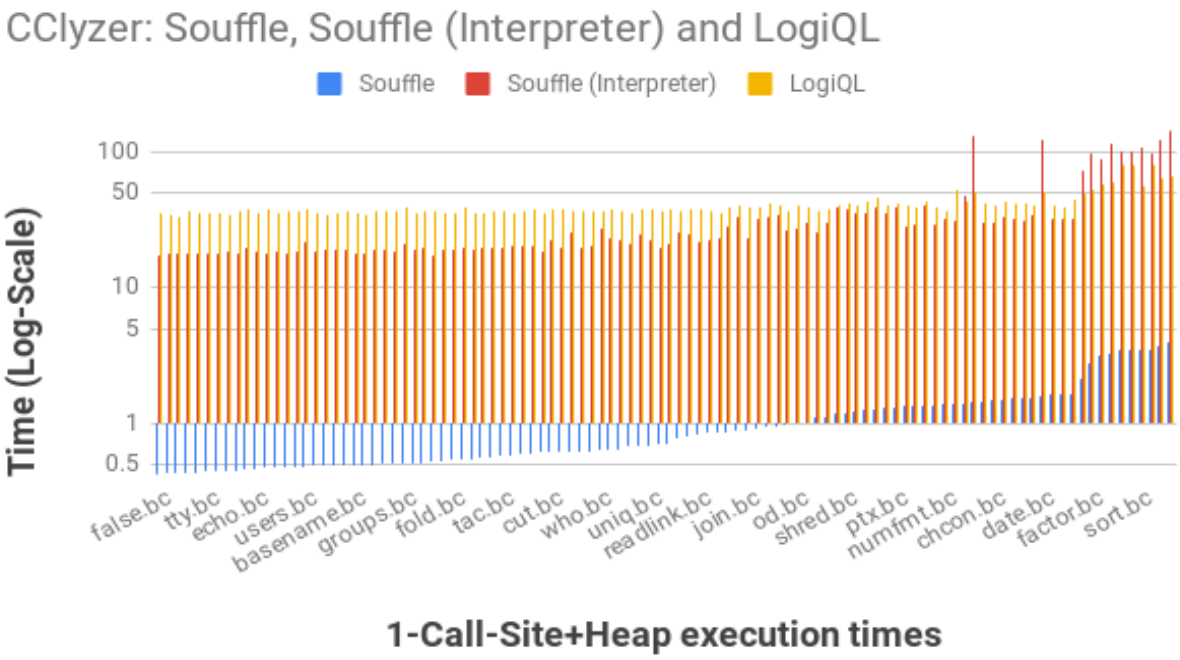

Figure 2: 1-Call-Site+Heap Sensitive Analysis running times (sec.)

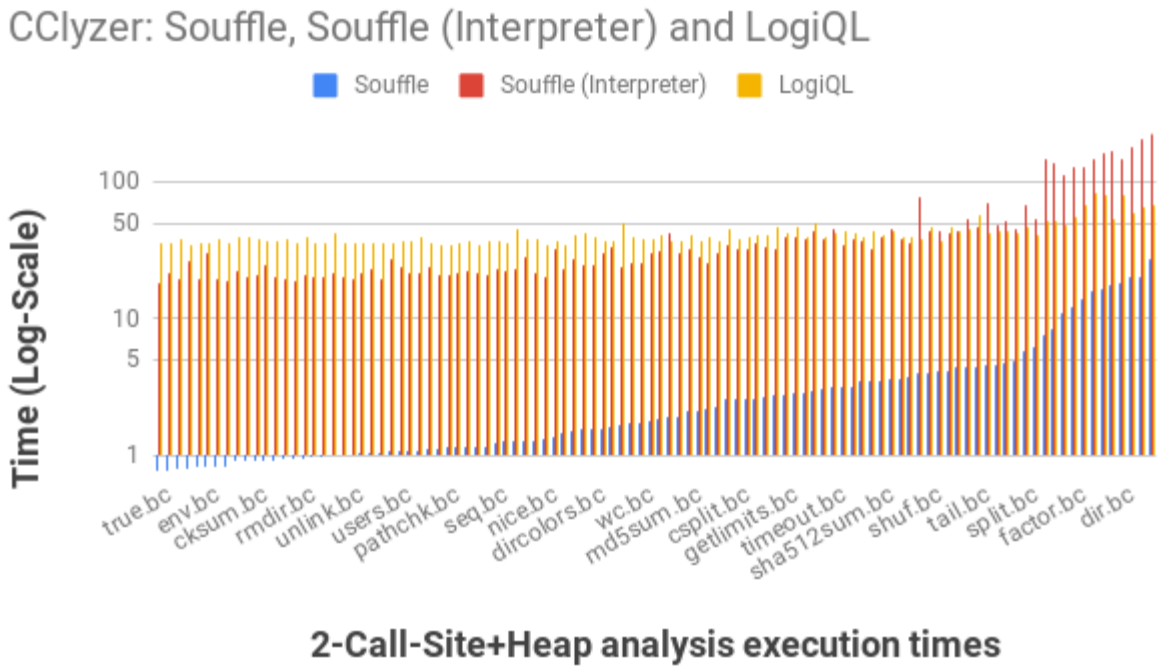

Figure 3: 2-Call-Site+Heap Sensitive Analysis running times (sec.)

The interpreter, although being close to the initial implementation, is not a consistent competitor as it performs in a negative manner for heavier workloads. On the other hand, the Souffle engine achieves an average of 95% acceleration in comparison to CClyzer's initial LogicBlox implementation.

However, we aim to achieve higher performance in terms of execution time, taking advantage of parallelism. To demonstrate this, we have to perform our analysis on a heavier workload. For that purpose, we compile a subset of Google's open-source

Chromium-web-browser source code in LLVM bitcode, in order to evaluate how parallelism affects our analysis' performance.
In the following section, we compare execution time for 1,2,4,8 and 16 threads on the same machine for all context variations.

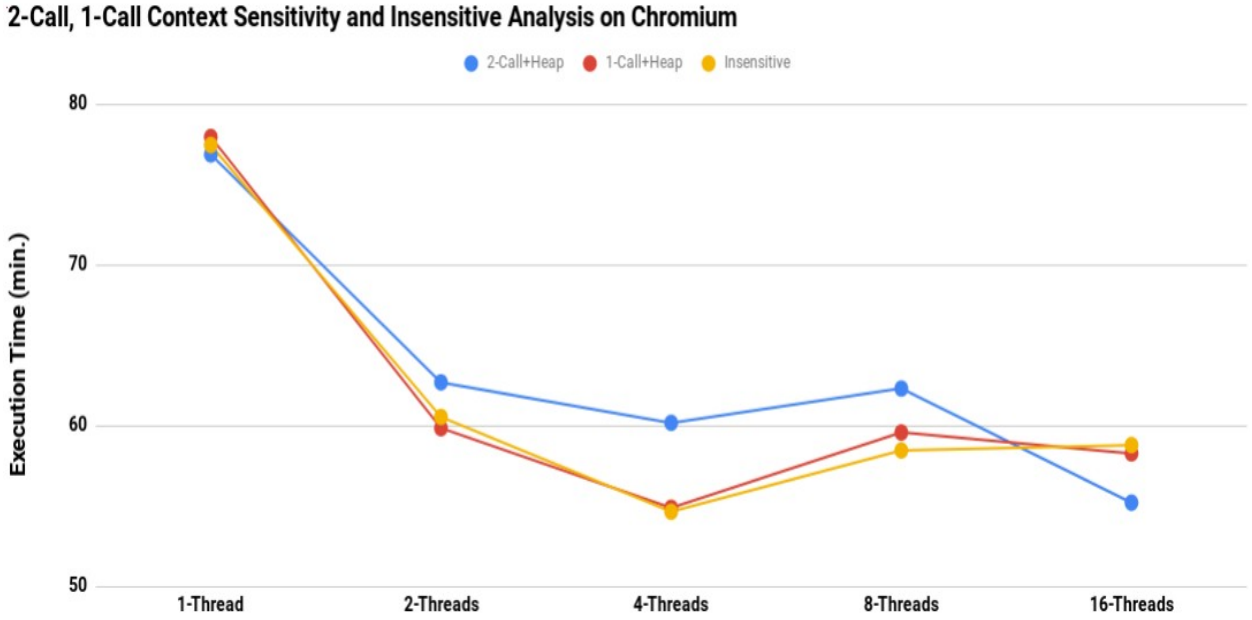

Figure 4: Running Time on different # of Threads for each Context-Sensitivity Variant (min.)

Table 1: Running Time on different # of Threads for each Context-Sensitivity Variant

Chromium					
(in minutes)	1-Thread	2-Threads	4-Threads	8-Threads	16-Threads
2-Call+Heap	76	62	60	62	55
1-Call+Heap	77	59	54	59	58
Insensitive	77	60	54	58	58

Here, our running time reaches its lowest point in the 4-thread execution with an acceleration of 29.87% in comparison to single-thread execution.

4.2 Discussion

Regarding our last comparison and differences between the two dialects and engines, there are some factors we are obliged to reference:

- Tweaking context sensitivity in each implementation requires it to be compiled again. This time has been significantly reduced in recent updates of Souffle but still holds a constant overhead of 3 to 5 minutes over the LogicBlox engine. There is a point to be made in also comparing compilation times between the two engines but already compiled analyses can be reused for any dataset without

any modifications. For the majority of use cases where the analysis is run for multiple datasets compilation is not considered a relevant variable.

- Populating EDB facts work differently for each engine. While Souffle directly imports facts from a file-based format, LogicBlox creates and populates a fully functional database. Each engine then exports their results in the same format. Because LogicBlox acts as a database that stores all intermediate relations we consider it fair for our Souffle implementation to also export all its relations, assuring identical results. Depending on the intended use of these results one may choose to prefer the database format of LogicBlox or the additionally reduced execution time of Souffle by exporting only facts to be examined.
- LogicBlox forces a strict type-system with dynamic tagging of types during its execution. In order for LogiQL to support constructors with newly created entities this process is essential for type inference. In Souffle, explicitly declaring types and their creation logic along with type checking during compilation, renders the need for such a system obsolete in the use case of our analysis.
- In its core, Souffle proves to be a fierce competitor to the LogicBlox Datalog engine, with up to 95% acceleration of run-time for our analysis with an additional 29% speed-up when taking advantage of its built-in optimized parallelism mechanism.

5. CONCLUSION

As static program analyses create more and more opportunities for automating the process of code optimization, it is imperative to search for solutions that offer acceleration of this process. Considering the vast and continuous growth of source code a modern application goes under, such analyses constitute valuable assets in the recognition of thresholds in compilation, execution and security of a modern application. Souffle is an engine that proves that “less is more” using its efficient declarative dialect, multiple optimization configurations, continuous evolution and adaptability to encourage building more complex analyses, while discarding processes that may require redundant computing effort. At the same time, by taking advantage of parallelism, it promises to make use of modern computer systems in their full potential. We believe that CClyzer is a tool that may still profit a great degree from further research of the engine’s available and future optimization features.

ABBREVIATIONS - ACRONYMS

EDB	Extensional Database
IDB	Intensional Database
LLVM	originally: Low Level Virtual Machine, no longer an acronym
LogiQL	Logic Query Language
Souffle	Systematic Ontological Undiscovered Fact Finding Logic Engine

REFERENCES

[1] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for C and C++. In Proc. of the 23rd International Symp. on Static Analysis, 2016.

[2] Y. Smaragdakis, G. Balatsouras. Pointer Analysis. Foundations and Trends® in Programming Languages 2015.

[3] Souffle Documentation, https://souffle-lang.github.io

[4] Dr. Terry Halpin and Dr. Spencer Rugaber. LogiQL: A Query Language for Smart Databases, Copyright © 2014 LogicBlox, Inc.

[5] T. Antoniadis, K. Triantafyllou, Y. Smaragdakis. Porting Doop to Souffle: A Tale of Inter-Engine Portability for Datalog-Based Analyses. In Proc. of SOAP, 2017.