On Fast Large-Scale Program Analysis in Datalog

Bernhard Scholz
Oracle Labs, Australia
bernhard.scholz@sydney.edu.au

Herbert Jordan
Oracle Labs, Australia
herbert@dps.uibk.ac.at

Pavle Subotić
University College London, UK
pavle.subotic@ucl.ac.uk

Till Westmann
Oracle Labs, USA
till@westmann.org

Abstract

Designing and crafting a static program analysis is challenging due to the complexity of the task at hand. Among the challenges are modelling the semantics of the input language, finding suitable abstractions for the analysis, and handwriting efficient code for the analysis in a traditional imperative language such as C++. Hence, the development of static program analysis tools is costly in terms of development time and resources for real world languages. To overcome, or at least alleviate the costs of developing a static program analysis, Datalog has been proposed as a domain specific language (DSL). With Datalog, a designer expresses a static program analysis in the form of a logical specification. While a domain specific language approach aids in the ease of development of program analyses, it is commonly accepted that such an approach has worse runtime performance than handcrafted static analysis tools.

In this work, we introduce a new program synthesis methodology for Datalog specifications to produce highly efficient monolithic C++ analyzers. The synthesis technique requires the re-interpretation of the semi-naïve evaluation as a scaffolding for translation using partial evaluation. To achieve high-performance, we employ staged-compilation techniques and specialize the underlying relational data structures for a given Datalog specification. Experimentation on benchmarks for large-scale program analysis validates the superior performance of our approach over available Datalog tools and demonstrates our competitiveness with state-of-the-art handcrafted tools.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying, Verifying and Reasoning about Programs; H.2.4 [*Information Systems*]: Systems–Query processing, Rule-based databases; D.3.4 [*Programming Languages*]: Processors–compilers

General Terms Languages, Performance

Keywords Static Program Analysis, Datalog, Program Synthesis, Compiler

1. Introduction

Program analyses are difficult to design and implement for real-world programming languages. The challenges range from faithfully modelling the semantics of the input programs, to finding sufficiently precise abstractions, to crafting a static program analysis resulting in thousands of lines of code in a traditional programming language. To overcome or at least alleviate the challenges in program analysis, Datalog [**? ? ? ? ? ? ? ? ? ? ?**] has been employed as a domain specific language (DSL). Program analyses can be expressed concisely in Datalog reducing the complexity and making the program analyses uniform such that they become interoperable with each other. Note that Datalog has received renewed interest in various computer science communities beside static program analysis, including information extraction, networking, security, and cloud computing as a DSL and a rapid-prototyping tool [**?**].

Datalog bridges the gap between specification and implementation, i.e., a programmer specifies a problem declaratively rather than describing it step-by-step, imperatively. A Datalog engine executes the specification for a set of input relations (also known as the extensional database) and produces an output relation for a query. There is a cornucopia of Datalog engines available. The most recent Datalog engines used in program analysis are bddbddb, μZ, and LogicBlox. The aforementioned engines use fast/compressed data structures and have pushed the boundaries of what previous logic-based analysis could achieve [**? ?**].

Despite these advances, the vast majority of the state-of-the-art analysis tools including points-to analysis [**?**] are developed manually, requiring typically thousands of lines of code. This is mainly due to program analyses specified in a declarative fashion still do not scale well for large-scale input programs with millions of program variables. Consequently, the question arises why there is a performance gap between modern Datalog engines and hand-written code, although modern engines use efficient evaluation techniques and sophisticated data structures for their evaluation. To answer this question, we state the research hypothesis that current Datalog engines do not adapt their evaluation according to their input specification.

In this work, we introduce a new angle to the problem: We perceive the problem of executing Datalog as a *program synthesis* problem whose goal is to automatically construct a program that satisfies the input specification. That is, we instantiate an analysis tool given a declarative Datalog specification of an analysis. While efficient synthesis from high-level to low-level imperative code has

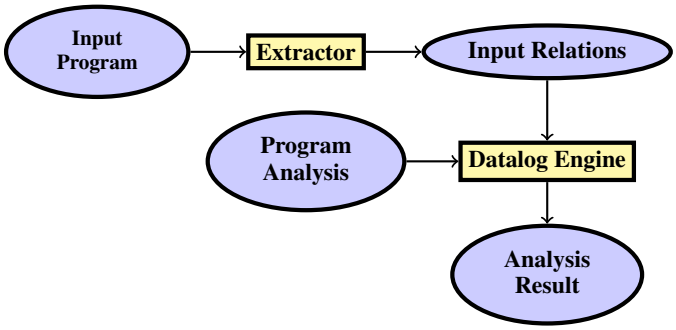

Figure 1. Typical Program Analysis in Datalog using an Extractor

received attention in the past [**?**], the intricacies of synthesizing Datalog to low-level imperative code has not received much attention, and general synthesis techniques cannot easily be adopted for Datalog evaluation.

We introduce a novel technique that goes beyond straight forward translations. Our synthesis approach is an adaptation of the seminaïve evaluation commonly used to evaluate Datalog programs. However, our technique addresses several key aspects of generating low-level imperative code from Datalog. Such aspects include the generation of specialized work-list algorithms for computing fixed-points, generation of indices for faster data querying and specialized data structures, among others. In this paper we show that Datalog-based analysis can be just as fast and scalable as hand crafted tools. The key insight is that the conversion from Datalog to low-level code must be done with the specifics of Datalog in mind. Our experimental results demonstrate the effectiveness of our technique. We are able to perform very large scale program analyses (millions of variables, hundreds of attributes, billions of tuples) at a fraction of the time compared to existing Datalog engines while being on par with the state-of-the-art in hand crafted tools.

We aim to give an insight to future designers and implementors of Datalog engines how to better optimize performance. While our focus in this paper is on program analyzers, we believe that our approach can be used to synthesize efficient general programs from Datalog. Our main contributions are:

- introducing a method of efficient program synthesis from Datalog analysis specifications
- specialization techniques for efficient parallelization and adaptive data structures
- providing experimental results including a points-to analysis written in Datalog that analyzes the OpenJDK[1] library with millions of variables in under a minute

The paper is organized as follows. In Section **??** we motivate our work by illustrating the performance gap between an analysis processed by existing Datalog engine and an equivalent, handwritten analysis. Sections **??**-**??** introduce our new program synthesis techniques to generate from a Datalog input specification executable OpenMP/C++ code. Section **??** briefly covers the implementation of our approach within the Soufflé Datalog engine, and we demonstrate the effectiveness of our new approach. We survey the related work in Section **??** and draw relevant conclusions in Section **??**.

2. Motivation

An example architecture of a Datalog-based program analysis framework is shown in Figure **??**. An extractor translates the input program to a collection of relations describing the relevant semantics of the input program. The input relations of a Datalog program are referred to as Extensional Database (EDB) in Datalog terminology. The program analysis itself is expressed in form of rules that is referred to as Intensional Database (IDB).

Consider the following, frequently occurring Datalog program where the IDB consists of the following rules,

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y),edge(Y,Z).
```

and the query is ?− path(X,Y). This program computes the transitive closure *path* of a given relation *edge* as EDB. Such an analysis may, for instance, be utilized to obtain a list of all reachable functions within a call graph. A slightly extended variation also provides the foundation for the widely applicable points-to analysis forming the foundation of a large variety of static program analysis. The given query can be converted into C++[2] within a few minutes, solely relying on constructs from the STL library. The central element is outlined in the following code snippet:

```
using Tuple = std::array<int,2>;
using Relation = std::set<Tuple>;

Relation edge, tc;

// fill edge relation
edge = someSource();

// eval first rule
tc = edge;

// eval second rule
auto delta = tc;
while(!delta.empty()) {
    // compute new delta
    Relation nDelta;
    for(const auto& t1 : delta) {
        auto a = edge.lower_bound({t1[1],0});
        auto b = edge.upper_bound({t1[1]+1,0});
        for(auto it = a; it != b; ++it) {
            auto& t2 = *it;
            Tuple tr({t1[0],t2[1]});
            if (!contains(tc,tr)) {
                nDelta.insert(tr);
            }
        }
    }
    // insert delta
    tc.insert(nDelta.begin(),nDelta.end());
    // swap data
    delta.swap(nDelta);
}
```

It is crucial for the performance of the C++ code, that entries in the edge relation are implicitly sorted enabling a range query for fast access. When compiling the given code fragment with GCC 4.9.3 and applying it on a graph with 1.000 vertices connected by 10.000 random edges, the program takes ≈ 2.0s and 91MB to compute all paths in the graph.

Executing the same problem on various Datalog engines takes ≈ 6.5s / 30MB (bddbddb), ≈ 340s / 1667MB (μZ), and ≈ 12.2s / 126.9MB (a SQLite based solver) on the same hardware (see Section **??**). The outcome of this simple experiment illustrates the significant performance gap between state-of-the-art Datalog engines and hand-crafted code.

[1] Java, JDK, and OpenJDK are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

[2] when referring to C++ we refer to the C++11 standard

While it may be still viable to manually craft code for small problems as outlined in the motivating problem, for a large scale, fully featured analysis instance this becomes inviable. A real world example may be comprised of hundreds of relations interlinked via numerous, recursive references, making it an extremely tedious and highly error prone programming task to manually write the analysis from scratch in an efficient way.

To overcome or at least alleviate the performance gap between state-of-the-art Datalog engines and handcrafted code, we propose a program synthesis approach to translate Datalog to specialized C++ code.

3. Framework Overview

Our framework translates a Datalog program to an executable that (1) adheres to the Datalog specification and (2) is highly optimized. We perform the program synthesis of Datalog specifications in four major stages: The first stage translates the declarative Datalog program to an abstract syntax tree. The second stage translates the abstract syntax tree to an abstract machine called the Relational Algebra Machine (RAM). The intermediate representation of the RAM features a concise set of relational algebra expressions, relation management statements, and control flow constructs with parallelism. The third stage translates the RAM program to C++. In the final stage, the C++ program is translated to a binary executable.

The architecture of our program synthesis framework for Datalog is depicted in Figure **??**. The four stages have different intermediate languages for representing the input program. In each stage, optimizations are performed to achieve a highly-efficient binary that is able to perform computations of large relations in parallel.

In the first stage, the Datalog specification is parsed and translated to an abstract syntax tree (AST). Semantic checks are conducted in this translational step including right usage of relation symbols, type checks of proper use of variables, and checks for cyclic negations among many other semantic checks. After the semantic checks, optimizations are performed. Those AST optimizations can also be perceived as a source-to-source translation, i.e., the input program is transformed to a more efficient input program (although represented as AST). The optimizations include:

- *constant propagation*, i.e., the forwarding of constant values within and among rules,
- *alias elimination*, i.e., the unification of variables according to equality constraints imposed by the user,
- *rule elimination*, i.e., the elimination of rules that do not contribute when atoms in their bodies refer to empty relations, and
- *relation elimination*, i.e., the elimination of entire relations and their rules, if they do not contribute towards the output.

In the second stage, the AST of the declarative input program is translated to an imperative relational description before it can be converted to C++. We introduce an abstract machine called the "Relational Algebra Machine" (RAM) for this purpose. The intermediate representation of the RAM constitutes a concise, imperative language providing a small set of simple control statements as well as operators and expressions to manipulate the state of relations. The lowering from a declarative Datalog program to an imperative relational algebra machine program is performed by re-interpreting the semi-naïve evaluation as a translation scheme [**?**]. The RAM representation of an input program offers ample opportunities to apply optimizations. Unlike on the AST level, mid-level optimizations may target details of the evaluation process not visible in the declarative specification. Among the most important mid-level optimizations is the conversion of simple traversal over relations to range queries and the associated problem of finding appropriate indices to speed up the evaluation of rules in form of nested join loops in RAM. The RAM representation of an input program provides enough abstraction to perform specific optimizations such as predicate leveling [**?**] and other optimizations that would be too tedious at a low-level such as C++.

The third stage converts the RAM immediate representation to C++ code. The main challenge of this translational step is to generate efficient, high performance C++ code for processing and storing information of in-memory relations. In our framework, we obtain sufficient performance by heavy use of C++ templates tailored for the use of relational algebra operations and efficient data structures including various types of indices. For specific instances of relations and operations we permit customizations of the code in the templates to achieve maximal performance. Thus, essentially, a large part of the actual code generation is deferred to the last translation, i.e., the C++ compiler that translates the heavily templatized input program to an executable program. The technique of scripting the generation of code using C++ templates is also known as template meta-programming. For example, if an actual type of an object is known at compile-time the dynamic dispatch is converted to a static call improving vastly the performance of the compiler. In our specific use-case, our meta-programming becomes a partial evaluator that pushes computations from runtime to compile-time if they are static.

In the final stage, the resulting C++ code is compiled to a binary executable. The C++ compiler unfolds the template producing highly efficient assembly code that is specialized for a given input-program. Using C++ makes the Datalog compilation independent of the actual target architecture.

4. Synthesizing from Datalog

The second stage synthesizes an imperative program from the declarative Datalog program represented as an AST using a step-by-step relational description of the input program. As aforementioned, the intermediate representation is an abstract machine program for the Relational Algebra Machine, i.e., an abstract machine specifically designed for relational algebra. The input program represented as a RAM program is further optimized using the optimizing transformations.

4.1 Adapting Semi-Naïve Algorithm

For translating an AST to a RAM program, we use the semi-naïve evaluation technique. The semi-naïve evaluation is a bottom-up evaluation of a Datalog program, i.e., results are constructed from the facts to the goal. In contrast to Prolog, this is possible because the relations have a finite domain and they grow monotonically until a fixed-point is reached. Ullman has shown [**?**] that semi-naïve evaluation is an efficient evaluation strategy. There are various survey papers and books discussing the semi-naïve evaluation for evaluating Datalog programs including [**? ? ?**]. The semi-naïve evaluation can be seen as a fixed-point computation where relations are initially empty. Clauses of the Datalog program are interpreted as semantic equations and are applied in a Kleene-fixed-point style, until no more new knowledge can be gathered. The obtained fixed-point represents the smallest model-theoretical solution, which coincides with the solution of the logical Datalog query [**?**].

For program synthesis, the semi-naïve evaluation is specialized for a given set of rules. The specialization produces a series of fixed-point computations in form of a program. A different interpretation is that a "Futamura projection" [**? ?**] is used to obtain a specialized evaluation for a given IDB, i.e., the Semi-Naïve Evaluation [**?**] can be seen as an interpreter that is specialized with its "static" intensional database (i.e. the Datalog rules representing the program analysis). As an outcome of the Futamura projection, we receive a relational algebra machine program that expresses the computation

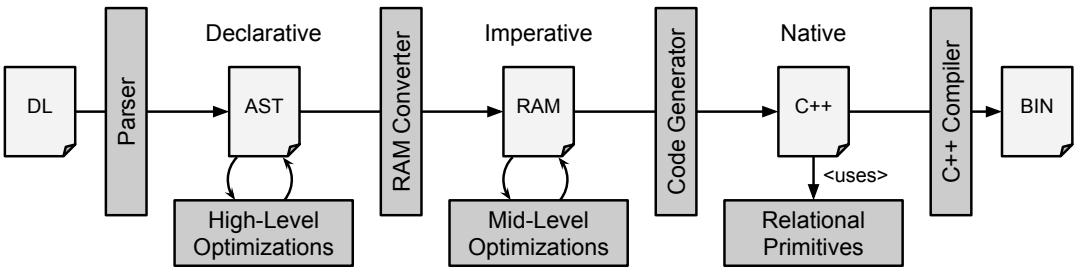

Figure 2. Staged Compilation Framework for Synthesizing C++ Programs from a Datalog Specification

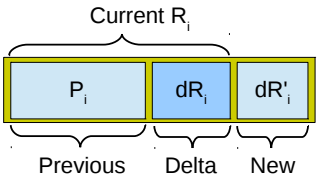

Figure 3. The semi-naïve evaluation splits recursively defined relations into subsets per fixed-point iteration called previous, current, delta, and new knowledge.

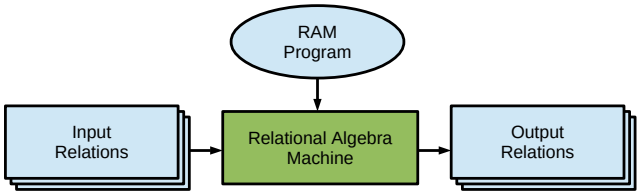

Figure 4. Relational Algebra Machine

of the program analysis as a series of fix-points performing relational algebra operations.

To avoid that all recursively defined relations are computed in a single fixed-point, the semi-naïve evaluator computes the data dependencies between relations in form of a precedence graph. Strongly-connected components (SCC) of the precedence graph resemble mutually recursive relations, and code for a fixed-point for these relations is generated. The SCC graph of the precedence graph represents a partial order that sequences the generated fixed-points and non-recursive rules such that the execution becomes efficient.

Consider the following Datalog program:

```
a(X) :- b(X), c(X).
b(1).
b(X) :- c(X), d(X).
c(2).
c(X) :- b(X), d(X).
d(3).
```

The fact $d(3)$ of relation can be safely evaluated before all others, i.e., number 3 is inserted into set d, since d is not depending on the value of any other relation. Also, a can only be computed once b and c have been computed, so the rules of relation a will be last to be evaluated. However, for b and c there is no total order since rules of both relations are mutually dependent on each other.

To avoid recurring computations, the semi-naïve evaluation scheme keeps track of the previous, current, delta and new knowledge of a recursively defined relation as depicted in Figure **??**. In this context, knowledge is defined as a set of tuples. In each iteration of the fixed-point iteration new knowledge is obtained, i.e., a new set of tuples is discovered for a recursively defined relation. The general observation is that only new knowledge in the previous iteration (i.e. delta knowledge) can generate new knowledge in the current iteration. Hence, for each iteration, relations are sliced into (1) current knowledge, which includes all the knowledge except the new knowledge of the current iteration, and (2) in the new knowledge gathered in the current iteration. With this partitioning of relations, the fixed-point will converge faster. In our semi-naïve code generation scheme we have for each recursively defined relation two helper relations: one relation that stores the new knowledge and another relation that stores the delta knowledge. The previous knowledge is deduced by set difference and is not stored explicitly.

For our example, the code generator emits code for a fixed-point calculation of relation b and c. The code fragment below outlines the code generated for the example above:

```
b = {1}; Δb = b;
c = {2}; Δc = c;
while (Δb ∪ Δc ≠ ∅) {
    new_b = (Δc ∩ d) \ b;
    new_c = (Δb ∩ d) \ c;
    b = b ∪ new_b;
    c = c ∪ new_c;
    Δb = new_b;
    Δc = new_c;
}
```

Inside a loop, new knowledge is obtained for relations b and c based on the Δb and Δc, i.e., the new knowledge of the previous iteration. The loop is executed until the Δ sets become empty. For more sophisticated queries, the notion of previous knowledge is required, i.e., the state of the relation in the previous iteration. A detailed description of the semi-naïve evaluation can be found in [**?**]. Once the relations b and c stabilize, a is computed by

```
a = b ∩ c
```

and an overall solution of the Datalog program is obtained.

The illustrated code generation pattern for semi-naïve evaluation can be generalized to an arbitrary Datalog program, and the conversion from the AST to the RAM is using semi-naïve evaluation as a code generator. For example, the motivating C++ example using STL containers has been constructed employing semi-naïve evaluation techniques.

4.2 Relational Algebra Machine

The Relational Algebra Machine (RAM) is an abstract machine that is used as a semantic model for evaluating translated input programs. The machine is specifically tailored for executing relational algebra programs that are produced by the semi-naïve evaluation. The RAM program contains relational algebra operations to compute results produced by clauses, relation management operations to keep track of previous, current and new knowledge in the semi-naïve evaluation,

and imperative constructs including statement composition for sequencing the operations, and loop construction with loop exit condition to express fixed-points computations for recursively-defined relations. We also extend the machine to execute statements in parallel via a parallel-statement. The parallel-statement processes a list of sub-statements in parallel and blocks until all of those have been completed. The static nature of a RAM program ensures that optimizations can be performed effectively, e.g., indexing, load balancing, etc.

The abstract machine operates solely on relations, which are sets of tuples. It has no explicit notion of variables and/or memory. The evaluation of a RAM program entails maintaining a collection of relations as a *state* for executing a RAM program. The relations are fixed throughout the execution of a RAM program, i.e., no new relation is added or deleted to/from the state whilst executing the program. However, the contents of a relation may change. Figure **??** depicts the execution model of a relational algebra machine program. There is a set of relations that the program operates on. Some of the relation are pre-loaded with data, e.g., the tuples defined by the facts in the original input program.

A RAM program is able to express nested join loops to execute Datalog clauses and the book-keeping portions of code for the fixed-point calculations. For example the rule of the previous code fragment

```
new_b = (Δc ∩ d) \ b;
```

is represented in RAM with a corresponding nested join loop similar to

```
for( x ∈ Δc )
  for ( y ∈ d )
    if ( x == y ∧ x ∉ b )
      b = b ∪ {x}
```

which will then be optimized by mid-level optimizations to

```
for( x ∈ Δc )
  if ( x ∈ d ∧ x ∉ b )
    b = b ∪ {x}
```

The RAM has the ability to express range queries for fast membership tests and simple searches. The RAM engine permits a wide range of optimizations including the selection of indices, and condition leveling, i.e., breaking conditions up and placing predicates to the most-outer loop in the loop-nest.

The statements of the relational algebra machine are listed below in BNF notation:

$$S \rightarrow \textbf{insert}\ O$$
$$O \rightarrow \textbf{search}\ R\ [\textbf{where}\ C]\ \textbf{do}\ O_1$$
$$O \rightarrow \textbf{project}\ (V_1, \ldots, V_k)\ \textbf{into}\ R_1$$
$$S \rightarrow \textbf{merge}\ R_1\ \textbf{into}\ R_2$$
$$S \rightarrow \textbf{swap}\ R_1\ \textbf{and}\ R_2$$
$$S \rightarrow \textbf{purge}\ R$$
$$S \rightarrow S_1; S_2$$
$$S \rightarrow \textbf{loop}\ S_1\ \textbf{endloop}$$
$$S \rightarrow \textbf{exit}\ C$$
$$S \rightarrow \textbf{par}\ S_1 || \ldots || S_k\ \textbf{endpar}$$

The statement **insert** represents a relational algebra statement performing the evaluation of a clause where O is a relational operation including cross product, selection and projection. Note the relational algebra machine has no high-level INSERT/SELECT statement as provided by the SQL standard, instead the nested join loop is expressed as cascaded searches followed by a project. The semantics of a search is the traversal over all tuples in relation R, and tests whether for a tuple the condition C holds. If it holds, the attached operation O_1 is executed recursively passing on the currently selected tuple of the traversal and the selected tuples of outer traversals. If the condition does not hold, the operation O_1 is skipped and the next tuple is assessed until the end of the relation is reached. The statement **merge** adds all tuples of relation R_1 to relation R_2. The statement **purge** deletes all tuples in relation R. The statement **swap** swaps the contents of two relations. Statements can be sequenced by semicolon by $S_1; S_2$ such that S_1 is executed prior to S_2. Loops are formed with statement **loop** ... **endloop** and can be terminated with **exit** C inside the loop body. The statement **par** $S_1 || \ldots || S_k$ **endpar** represents the parallel execution of statements $S_1, \ldots, S_k$. It continues executing until all statements $S_1, \ldots, S_k$ have been terminated. The sequence instruction and the parallel instruction implements a series-parallel based programming model. Note that the code generator is responsible for producing code that does not contain races since the consistency is not enforced by the abstract machine, i.e., a parallel section does not allow the parallel execution of two or more project statements that perform insertions on the same table.

We demonstrate the translation from AST to RAM using the example from Section **??**, which computes the transitive closure of the binary relation *edge* and stores the result in the relation *path*:

```
edge(1,2).
edge(2,3).
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y),edge(Y,Z).
```

The SCC graph of the example above contains two components. A non-recursive component that contains the relation *edge* and a recursive component *path*. The topological order will enforce the evaluation of relation *edge* before relation *path*. In the first step, the tuples of the facts are loaded in the relation *edge*. In the second step the path relation is computed recursively. The relaxed semi-naïve evaluation lowers the Datalog program above into the program as shown in Figure **??**. In the example, the rule

```
path(X,Y) :- edge(X,Y).
```

is non-recursive and is executed outside the fixpoint-loop. The tuples of the relation *path* are transferred to the relation *delta_path* for bootstrapping the fix-point loop. Inside the loop, the recursive rule

```
path(X,Z) :- path(X,Y),edge(Y,Z).
```

is translated to a nested join loop. The loop terminates if no new knowledge can be found. At the end of a loop iteration the tuples of *new_path* are transferred to *delta_path* for the next iteration.

4.3 Index Selection

Range queries are query operations on indices. Since indices are costly, one would ultimately desire a minimal set of indices for a given relation. To find a minimal set of indices, we employ a discrete optimization problem and have devised a solver for it.

Consider a relation r with 3 attributes X, Y, and Z that is used in the following rule:

```
a(X) :- b(X), r(X,Y,X), c(Y).
```

which searches for all values in the relation r where the first and the last component is of a value X stored in b and checks then whether the second component is also present in the relation c. The nested join loop of the loop may be the following one:

```
for( x ∈ b )
  for( (y,z,w) ∈ {(y,z,w) ∈ r | y = x ∧ w = x} )
    if ( z ∈ c ∧ x ∉ a )
      a = a ∪ {x}
```

```
// populate fact tuples
project (1, 2) into edge;
project (2, 3) into edge;

// rule: path(X,Y) :- edge(X,Y).
insert
 search edge do
  project (edge[0],edge[1]) into path;

// create delta knowledge for first iteration
merge path into delta_path;

// fixed-point loop
loop
 // reset helper relation
 purge new_path;
 // rule: path(X,Z) :- edge(X,Y), path(Y,Z).
 insert
  search edge do
   search delta_path where (
            (edge[1] = delta_path[0]) and
            ((edge[0],delta_path[1]) not in path)
         ) do
    project (edge[0],delta_path[1]) into new_path
        ;

 // fixpoint reached?
 exit counttuples(new_path) = 0;

 // book-keeping
 merge new_path into path;
 swap new_path and delta_path;

endloop
```

Figure 5. RAM program for Datalog program with helper relations *new_path* and *delta_path* that store the new knowledge of the current and previous iteration, respectively.

where the second loop is an application of a range query on the relation r. To effectively query all elements of r with a given value for the first attribute X and last attribute Z a corresponding index on the relation is required. For instance, an index sorting all elements according to the attribute order $X \prec Z$ would be suitable. So would be an index ordered by $Z \prec X$, $X \prec Z \prec Y$, or $Z \prec X \prec Y$. The latter two are even supporting an efficient query on all 3 attributes while only those starting with an X support queries on the X attribute only. This observation, that indices may be shared among several queries, is utilized by our index selection optimization. The transformation collects all search patterns for a relation that are used in range queries, and computes the minimal set of indices using a discrete optimization problem. The indices will then be created at the creation-time of a relation and are maintained throughout the lifetime of a relation.

5. Native Optimizations

In the third stage, a RAM program is translated to a templatized C++ program. The templatized C++ program realizes efficient relational algebra operations including emptiness checks, membership tests, scans, range queries, insertions, and union operations. As we show in Section **??** the most performance critical operations are the insertion and query operations. It is therefore of paramount importance to have an effective code generator using templates that generates efficient data structures for relations and their operations.

5.1 Data Structures

In our execution model we keep relations in memory. Given the increasing availability of computers with terabytes of memory, this is a viable option. There are many different data structures that can operate on relations stored in memory including Hashes, Trees, Tries, Bit-vectors, or Sorted lists. However, most of the aforementioned data structures are not suitable for very large relations. For example, Hashes, while being efficient for inserts and membership tests, do not provide acceptable range query support and the hash table size deteriorates the cache performance, and as a consequence the overall run-time. Sorted lists are expensive for insertions and Bit-vectors require large amounts of space, in particular for higher-dimensional tuples. In our experience, the only viable approaches to store very large relations are balanced search trees and Tries. For these membership tests and range queries exhibit a worst-case execution time of $\mathcal{O}(\log(N))$.

Among various types of balanced search trees, B-trees, which were originally designed for secondary storage data structures, are known to be the most memory efficient and cache effective data structures. Therefore, we employ in-memory B-trees as our primary data structure for storing very large relations to obtain performance. However, a reorganization of B-trees becomes costly for executing relational operations in parallel. To overcome this issue for some relations, we introduce an alternative datastructure that is based on geometrically encoded Tries. The choice of the data-structure is dependent on the relation and its use. Both implementations are interchangeable, since both data structures use the same API.

In the evaluation of the respective performance trade-offs between B-trees and Tries we discovered that in our use-cases, Tries provide better performance for relations with one or two attributes. We provide experimental evidence in Section **??** for the selection of the possible data structure. With a growing number of attributes, the Trie data structure becomes ineffective in terms of run-time and memory consumption, and the B-tree outperforms the trie data structure. From our empirical analysis, we derived the following decision table:

# of attributes	Number of Indices $0-1$	Number of Indices ≥ 2
0	flag	-
$1-2$	Trie	Trie
$3-5$	B-tree	B-tree
$6-n$	B-tree	blocked list + indirect B-tree index

Based on the number of attributes and the number of indices we are switching between B-tree and Trie which store the relation directly. This is also known as indexed-organized tables. For relations with many attributes, the table is stored in a blocked list and indices contain pointers pointing to the records in the list in order to save memory.

The data structures are written in a template meta-programming library that also provides the mechanisms to choose the right data structure for a given relations and hence simplifies the code generation.

5.2 Specialization

Besides the selection of the data structure, additional parameters enable the C++ template mechanisms to produce very specialized code for each individual relation within the resulting program. Those parameters include:

- *arity* – the number of attributes of the tuples to be stored in a relation
- *primary index order* – the order to be utilized for sorting attributes in the main data store of a relation; also the index utilized for membership tests;

- *secondary index orders* – a list of indices to be associated and automatically maintained for a relation; queries may select those instead of the primary index if more suitable

Those template parameters are utilized to synthesize, e.g., comparison operators and search operations on B-tree keys and nodes specifically tailored for the individual use cases. As a consequence, the optimizing C++ compiler is enabled to conduct more low-level code optimizations due to considerably reduced data dependent control flow decisions. Essentially, every relation and every index gets its very own, specialized implementation.

5.3 Parallelization

There are two elements to be consider to effectively utilize contemporary hardware: caches and parallel cores. The effective utilization of caches is achieved by choosing appropriate data structures. To harvest the computational power of parallel cores, the code generator issues parallel constructs.

A Datalog program provides ample of opportunities for parallelization. The most relevant code portions to parallelize are the executions of nested join loop. For instance, the loop nest

```
for( x ∈ b )
  for( (y,z,w) ∈ {(y,z,w) ∈ r | y = x ∧ w = x} )
    if ( z ∈ c ∧ x ∉ a )
      a = a ∪ {x}
```

can be parallelized by partitioning the relation b and distributing the partition among multiple, parallel resources. However, to be a valid transformation, all operations conducted within the loop nest have to be thread safe. Note that scanning, querying and checking for memberships are pure read-only operations which can always be processed safely in parallel. The only critical operation is the insertion of new values into a in the innermost loop. This update operation on the set-representation of a needs to be synchronized. However, the synchronization only needs to protect concurrent inserts. A protection against e.g., concurrent scan and insert operations is not necessary since such combinations cannot occur in a RAM program produced by the semi-naïve evaluation strategy.

To protect concurrent inserts for B-trees, several strategies are available. The simplest one is to protect concurrent insertion operations by locking the entire tree, thus sequentializing updates. Unfortunately, this also severely limits the parallel efficiency of the resulting code since due to lock contention, threads block each other in the execution of insert operations. Consequently, a locking strategy involving the underlying data structure on a finer granularity is required.

For Tries the synchronization operation for insertions can be implemented using atomic updates, thus realizing a lock-free data structure. Whenever a new node is inserted, a null-pointer somewhere in the structure will be atomically updated to point to the new node. If the update fails, the insertion procedure is simply re-started. This lead to a highly scalable parallel implementation.

For B-trees on the other hand, the synchronization is a bigger challenge since insertions are not restricted to updating a single memory location. In the general case, keys and child pointers need to be shifted and potentially parent nodes split and re-balanced. The application of a fine-grained read/write locking scheme protecting all the nodes potentially affected by an insert operation and releasing locks as early as possible provided acceptable scalability on desktop systems. However, on multi-socket server systems the continued exchange of updates on the lock associated to the root node over the inter-chip buses caused a severe slow-down in performance and scalability. As a result, even with the fine-grained locking parallelism on multi-socket systems did not provide any net gains in performance.

Tool	Time [s]	Memory [MB]
C++	2.0	91
bddbddb	6.5	30
μZ	340	1667
SQLite	12.2	126.9
Soufflé / B-tree (sequential)	1.26	25.6
Soufflé / B-tree (parallel)	0.42	26.3
Soufflé / Trie (sequential)	0.38	3.5
Soufflé / Trie (parallel)	0.12	4.5

Table 1. Comparison of Datalog evaluation tools for random-graph connectivity problem.

To overcome this limitation we adapted an optimistic locking schema from databases. In this approach, every node in the tree is annotated by a version number which will be updated upon every modification. When a thread is reading a node while navigating the B-tree during an insert operation, it is recording the version number before starting its operation and comparing it after determining the next node to navigate to. If the version number remained unchanged, it continues by navigating to the resolved node. However, if the version number changed, some other thread has modified the content of the processed node while the read operation was in progress. Thus, the obtained result may be wrong. To correct, the thread simply restarts the read operation on the same node again.

Compared to the fine-grained locking, the optimistic locking approach does not update any memory location (or lock state) when there are no conflicts – which is the case in the vast majority of node traversals. Thus, communication between sockets is significantly reduced, leading to largely superior parallel scalability compared to the fine-grained locking solution.

For our implementation we are utilizing the primitives provided by OpenMP to mark corresponding loops to be processed in parallel. Thus, the degree of parallelism can be controlled by the user.

6. Experimental Results

We have implemented our program synthesis techniques for Datalog within Soufflé– a high performance Datalog engine for large-scale program analysis. Soufflé has been designed for the highest performance and scalability. Currently, the Soufflé project is undergoing an open source licencing process and is expected to be released for general use in the first half of 2016. The following sub-sections provide experimental data regarding our system's performance characteristics compared to state-of-the-art Datalog engines that have been used for static program analysis in literature.

6.1 Transitive Closure

Our first experiment targets the evaluation of our motivating example introduced in Section **??**. In this example, the transitive closure of a random graph has been computed utilizing a variety of tools. The experiments have been conducted on a 8 core Intel i7-5820K CPU @ 3.30GHz. Table **??** summarizes the observed execution times and memory requirements.

The C++ entry in the table corresponds to the implementation of the transitive closure using standard C++ containers as outlined in the motivation section. The results of the bddbddb and μZ rows have been obtained utilizing the corresponding tools. The SQLite evaluation is based on the consecutive execution of SQL statements according to the semi-naïve evaluation scheme utilizing the in-memory relational database system SQLite. All three represent manifestations of distinct concepts for their internal data representation. However, all of them, some severely, fail to compete with the naïve C++ implementation in terms of execution time and only bddbddb manages to obtain the result by consuming less memory.

Tool	Time [hh:mm::ss]	Memory [GB]
bddbddb	0:30:00	5.7
μZ	DNF	DNF
SQLite	6:20:00	40.2
Soufflé (sequential)	0:01:15	7.5
Soufflé (parallel)	0:00:35	8.5

Table 2. Comparison of Datalog evaluation tools for a context-insensitive points-to analysis on the OpenJDK7 library.

Operation	% time	# calls
Inserts	$\approx 45\%$	$\approx 200G$
Membership Test	$\approx 35\%$	$\approx 180G$
Range Queries	$\approx 15\%$	$\approx 20M$
Scans	$\approx 5\%$	<inlined >
Rest	<1%	-

Table 3. Runtime Profile of points-to Analysis run on OpenJDK7

In comparison, our Soufflé engine is capable to improve upon the naïve C++ performance by decreasing the execution time by 40% and using less then $1/3$ of the memory requirements running in a sequential, B-tree based mode. By using our Trie-based data structure, an execution over 16x faster with less than $1/20$ of the memory requirements can be observed.

The significant execution time reduction of Soufflé compared to the naïve C++ version is based on the improved cache utilization due to the employment of cache friendly, low overhead data structures, the specialization of the involved data structures for the particular use case, and the employment of fine-grained parallelization to effectively and efficiently utilize concurrent hardware resources.

However, Soufflé's higher level optimizations, in particular it's automated index selection capability, is not coming into effect in this small transitive closure example. Neither is Soufflé's capability of managing relations comprising billions of tuples. For this, a large scale real-world analysis needs to be considered.

6.2 A Real-World Benchmark

A points-to analysis (which program variable is pointing to which object) is a core analysis for a variety of static program analysis, including, escape analysis, loop-dependency analysis and security analyses. The corresponding points-to analysis can be formulated as a Datalog query. To evaluate the various elements contributing to the performance of Soufflé we have evaluated the application of a context-insensitive version of the points-to analysis on the entire code base of the OpenJDK library containing 1.4M variables, 350K heap objects, 160K methods, 590K invocations and 17K types.

Table **??** summarizes the performance characteristics of various state-of-the-art tools for Datalog based program analysis for the given problem statement. All of them have been processing the same Datalog program comprising several dozen relations and rules producing $\approx$ $840M$ resulting tuples when being applied to the OpenJDK7 input set. The evaluations of the bddbddb, μZ and SQLite and based analysis have been conducted on a 8 core Intel Xeon E5-2690 v2 @ 3.0GHz, 128GB RAM server system due to resource and licencing constraints, while the Soufflé experiments have been conducted on a 4 core Intel i7-4790 CPU @ 3.6GHz, 32GB RAM desktop system. However, the huge performance gap between the various approaches are far beyond what can arise from the performance discrepancy of the hardware.

For μZ the size of the resulting Datalog query has been too large to obtain results within a reasonable time (DNF = did not finish). Also, the SQL engine based Datalog solver required a significant amount of computation time, rendering it practically unsuitable for the development of real-world, large scale analysis. bddbddb could handle the query within much more reasonable time scales. For this real-world benchmark Soufflé is capable of computing the desired result more than 34x faster then the best state-of-the-art solver – a factor that moves the development of more sophisticated large-scale static programming analysis from the realm of academic exercises into practical reality.

6.2.1 Performance Breakdown

During the course of the development of Soufflé we have used the context insensitive version of the points-to analysis as a benchmark to trace the gradual improvement steps of our system. Thus, we can provide additional insight on the specific gains that can be obtained by integrating various optimizations.

Table **??** summarizes profiling data collected from the execution of a single Soufflé based run of the points-to analysis on the OpenJDK7 dataset. As can be observed, the vast majority of time ($> 99\%$) is spent on operations on sets. In particular, insert operations and membership tests, which are triggered several billion times, account for $\approx$ 80% of the overall execution time. Thus, the corresponding operations have been a highly valuable target for optimizations. Unfortunately, each evaluation only consumes fractions of a micro second, leading to the necessity of resorting to a variety of low-level optimization steps in the implementation as well as modifications improving the cache utilization.

Figure **??** outlines the series of major development steps our system was undergoing, visualizing the gradual reduction of execution time and memory consumption.

The base line for our comparison is given on the left by the initial version of the C++ code Soufflé has been producing for computing Datalog queries. This initial version required $\approx 850s$ and roughly 17GB of memory, which already constituted a vast improvement over preexisting solvers. However, the subsequent improvement steps could reduce the execution time by an additional 93% and the memory consumption by up to 59% for the given real-world benchmark.

The first optimization was reducing the management overhead by reducing the number of maintained indices for each relation to a minimum which reduced the execution time by $\approx 200s$. The next major optimization refers to storing tuples in the index structure if only a single index is required for a relation. This improved cache efficiency, thus reduced the execution time by $\approx$ 138s and the memory consumption by $\approx 7GB$. It was followed by a switch to a custom B-tree implementation ($57s$ improvement) followed by a sequence of low level optimizations including the manual re-write of recursive code into iterative code ($35s$), and the reduction of control flow dependencies – thus pipeline hazards – in the tuple comparison operators ($33s$). To exploit the temporal locality of resulting query operations, which frequently query sequences of nearby elements in the data relations, we remember last accessed nodes as hints for future operations. This resulted in avoiding redundant lookups ($\approx$ 90% hit rate) and a run-time reduction of another $\approx$ 122s. Next, we customized the implementation of the binary search operation frequently performed by B-tree operations. The effect was an improved effectiveness of operations and a run-time reduction of additional 50s. The utilization of a 3-way comparison instead of consecutive binary comparison operators, the improvement of the set-merge operation by ensuring that the smaller is added to the larger set, the adaptation of node-split strategies to increase the fill rate of nodes, and the utilization of locally available delta-sets as a filter for membership tests decreased the run-time by an additional $\approx 97s$. The introduction of our Trie based data structure for binary

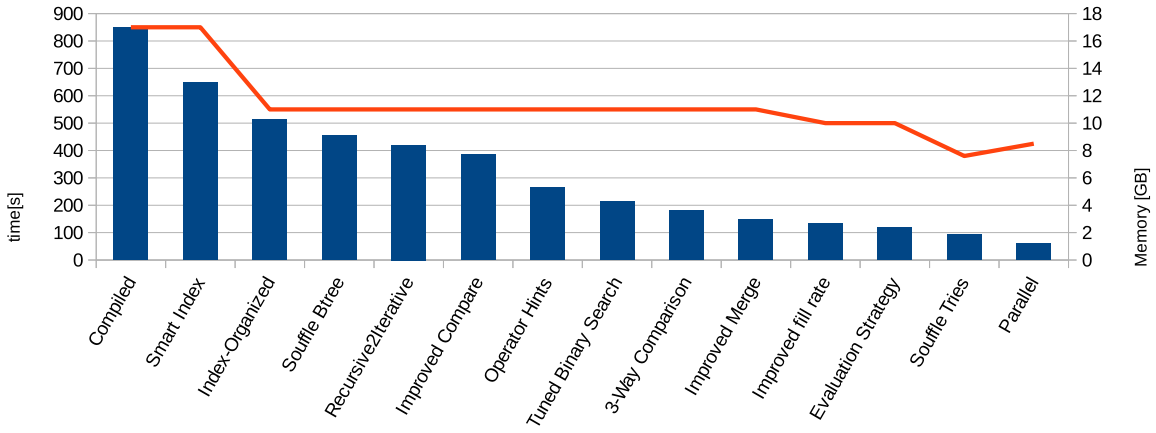

Figure 6. Breakdown of Performance Improvements

Step	Absolute Time [s]	Relative Reduction	Relative Speedup
semi-naïve on SQL engine	22800	-	-
C++ Encoding	850	-96%	$26.8\times$
Index Consolidation	650	-24%	$1.3\times$
A&D Tuning	75	-88%	$8.7\times$
Parallelization	35	-53%	$2.1\times$
Total	35	-99.8%	$651\times$

Table 4. Performance Improvement Breakdown for context-insensitive points-to analysis on OpenJDK7

relations reduced the execution time by an additional $22s$ and the memory usage to $7.5GB$. Finally, by parallelizing our engine we managed to reduce the computation time to less than a minute on our test system at the expense of sightly increased memory requirements.

Table **??** summarizes the incremental gains achieved over a SQL based datalog engine by encoding queries in C++, minimizing the number of utilized indices, conducting the manual tuning steps on basic algorithms and data structures as outlined above, and parallelization. Unfortunately, due to interdependencies between the various optimization steps outlined in this section, their individual impact can not be evaluated in more detail.

6.2.2 Comparison to Manual Implementation

Within recent work, the points-to problem over the OpenJDK library has been investigated in detail and a specialized, graph based algorithm for its efficient computation has been devised [**?**] in Java. In particular, the proposed solution comprises specialized data structures to effectively represent and compute the points-to relation. The work has the groundbreaking capability of obtaining the analysis results for the OpenJDK library in under a minute. With our Datalog engine the same result on the same dataset can be obtained utilizing a general purpose analysis infrastructure within 35s on a commodity desktop system. This result provides an indication on the competitiveness of our Datalog engine in regard to manually encoded static program analysis.

6.3 Large-Scale Analysis

Our last experiment evaluates Soufflé's capability of providing the computational framework exceeding the practical capabilities of state-of-the-art solvers when processing even more sophisticated large-scale analysis. To that end, we have been processing a *context-sensitive* points-to analysis on OpenJDK7 build 147. The context-sensitive analysis is a 2-Object-1-Heap points-to analysis [**?**] using an open-world abstraction [**?**]. The evaluation has been conducted on a 8 core Intel Xeon E5-2690 v2 @ 3.0GHz server system. Table **??** summarizes the obtained results.

Tool	Time [hh:mm::ss]	Memory [GB]
bddbddb	DNF	DNF
μZ	DNF	DNF
SQLite	DNF	DNF
Soufflé (parallel, 8 cores)	6:44:08	825.77GB

Table 5. Comparison of Datalog evaluation tools for a context-sensitive points-to analysis on the OpenJDK7 library.

Only Soufflé has been able to cope with the large-scale program analysis problem for analyzing context-sensitive points-to on OpenJDK.

7. Related Work

Datalog Engines. In this section we describe briefly survey state-of-the-art Datalog engines and their use in the context of static program analysis. bddbddb [**?**] is one of the most popular Datalog engines for static program analysis. The main challenge in using bddbddb for large systems relates to the issue of variable ordering. As it is uses BDDs (binary decision diagrams) as the underlying structure, choosing the right ordering is of paramount importance. Otherwise, the analysis does not terminate within reasonable bounds. In our experimentation the default variable ordering did not work for the JDK but after significant exploration we were able to get it working. However, this variable ordering was not useful for the analysis of a different version of the JDK. Such repeated exploration to find suitable variable orderings is too time consuming for bddbddb to be useful in our context. μZ [**?**] is another tool that does very well on small examples. However, the tool was unable to handle the large data sets generated during the analysis of the JDK. However, our approach still improves on its performance. A common difference between the aforementioned Datalog engines and Soufflé is that they perform Datalog evaluation whereas we use Datalog as a specification to synthesize a C++ program.

In [**?**] and [**?**], source-to-source translators from Datalog to SQL were introduced. Unfortunately, current relational database management systems cannot cope with the vast amounts of data and

complex queries that arise translating Datalog to SQL. Other systems such as IRIS [**?**] and DLV [**?**] provide support for Datalog execution. Both of them are bottom-up rule inference engines. However, they cannot be used as a stand-alone system. They provide the basic knowledge base component and the actual application needs to be written in a language like C++. Datalog Education System [**?**] is a deductive database system that supports querying via both Datalog and SQL. Their focus is to support SQL queries in Datalog and thus translate SQL into Datalog. Socialite [**?**] provides extensions to Datalog to facilitate parallel execution. The aim is to speed up various graph algorithms and hence provide support for features such as aggregation. The programmer needs to provide suitable annotations to enable effective parallelization on distributed systems. None of these tools are suitable for static program analysis either due to their design targeting different application domains or will have performance issues tackling large data with billion tuples in relations.

Liu and Stoller [**?**] describe a general method for transforming Datalog rules to SETL programs. Their focus is on guaranteed worst-case time and space complexities. They use a mixture of arrays and linked list to manipulate the various sets. While this is useful in guaranteeing worst-case complexities their experiments are on relatively small data sets. Thus, it is not clear if their approach can handle large data sets. There are other approaches to implementing Datalog engines using GPUs [**?**] that harness the parallel capabilities of accelerators. In their work, tables may store the same tuple several times, and enforcing a set constraint at a later stage becomes costly, dominating the overall execution time. The duplication of tuples depletes the GPU memory quickly, and memory limitations of contemporary GPUs just amplify the short-coming of their approach for large-scale program analysis.

Synthesis of Analyzers. In our context, program synthesis refers to the classical notion for it [**?**] i.e., constructing an executable program (i.e., program analyzer) from a logical specification (i.e., in Datalog). We refer the reader to [**?**] for a survey of program synthesis techniques and uses. While our framework can also be used to generate C++ programs from Datalog specifications, our focus in this paper is efficient synthesis of program analyzers, i.e, we generate C++ programs that take a program as a set of relations and produce analysis results in output relations. Several frameworks have been cast as a synthesis of analyzers and/or verifiers e.g.,[**?** **?**], and to a lesser extent [**?** **?** **?**]. While our approach shares similarities of specifying the analysis in a logical specification (Datalog Horn clauses) we generate a stand-alone C++ analysis tool rather than solving clauses within the framework/engine itself. As shown in our experiments, this results in significant performance gains. The approach in [**?**], like us, uses partial evaluation of Datalog to improve performance. Additionally, several compilers perform efficient code generation using synthesis techniques [**?** **?**]. This body of work, like our technique, synthesizes efficient code form a logical specification; unlike our work, these approaches generate general programs optimized at the assembly level, where as we generate analysis tools optimized at the C++ level, adhering to a Datalog specification.

8. Conclusion

We have presented a Datalog-based analysis framework that instead of evaluating Datalog, uses the semi-naïve algorithm as a synthesis scaffold to produce analyzer instances. Our technique converts Datalog specifications to efficient, parallel C++ code. As a result, we are able to analyze very large code bases and perform analyses considerably more efficiently then the available state-of-the-art Datalog engines. While our work focuses on static analyzers, we believe that our approach can also be used as an efficient general purpose engine for compiling Datalog to efficient C++ translations. Our work has been realized in the Soufflé framework which is currently undergoing an open-source process and is estimated to be publicly available in the first half of 2016.

Acknowledgments

We would like to thank our colleagues at Oracle Labs, Brisbane and Nicholas Hollingum.