An Efficient Interpreter for Datalog by De-specializing Relations

Xiaowen Hu
School of Computer Science
The University of Sydney
Australia
xihu5895@uni.sydney.edu.au

David Zhao
School of Computer Science
The University of Sydney
Australia
dzha3983@uni.sydney.edu.au

Herbert Jordan
University of Innsbruck
Austria
herbert.jordan@uibk.ac.at

Bernhard Scholz
School of Computer Science
The University of Sydney
Australia
bernhard.scholz@sydney.edu.au

Abstract

Datalog is becoming increasingly popular as a standard tool for a variety of use cases. Modern Datalog engines can achieve high performance by specializing data structures for relational operations. For example, the Datalog engine Soufflé achieves high performance with a synthesizer that specializes data structures for relations. However, the synthesizer cannot always be deployed, and a fast interpreter is required.

This work introduces the design and implementation of the Soufflé Tree Interpreter (STI). Key for the performance of the STI is the support for fast operations on *relations*. We obtain fast operations by *de-specializing* data structures so that they can work in a virtual execution environment. Our new interpreter achieves a competitive performance slowdown between 1.32 and 5.67× when compared to synthesized code. If compile time overheads of the synthesizer are also considered, the interpreter can be 6.46× faster on average for the first run.

CCS Concepts: • **Software and its engineering** → **Interpreters**.

Keywords: Interpreter Implementation, Datalog Engine, Static Data Structure

ACM Reference Format:
Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. An Efficient Interpreter for Datalog by De-specializing Relations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454070

1 Introduction

Logic programming languages, such as Datalog, are gaining in popularity for a variety of analysis problems due to their succinctness and ease of use. In recent years, Datalog has been extended in its expressiveness (e.g., functors, aggregates, records, lattices, etc.) and used for various applications, including static program analysis in Java [4] and Tensor Flow programs [34], security-oriented analysis in Ethereum Virtual Machine [24] and network analysis in Amazon Web Services [7].

For these large-scale applications, the relations of a Datalog program may contain up to billions of tuples. Hence, it is performance-critical to efficiently represent and interact with relations. Soufflé [28, 47] is a Datalog engine that uses in-memory data structures for relations and has high performance due to its synthesizer. The synthesizer compiles a Datalog program into highly optimized and parallel C++ code, and can achieve performance comparable to optimized, hand-crafted code.

However, the synthesizer is not always the preferred mode of execution. For example, Datalog programs may be executed in a Cloud environment that cannot run C++ compilation tools (such as the Amazon Lambda Service in some applications [7]). Other scenarios include the development and debugging of Datalog applications, where re-compilation for small program changes becomes too tedious. For such scenarios, a fast interpreter for Datalog programs is paramount.

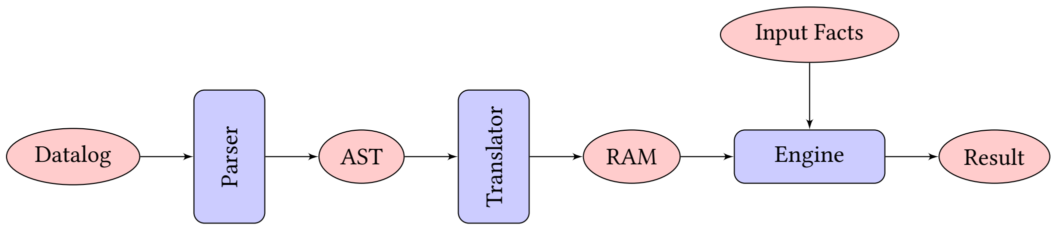


Figure 1. Execution model of Soufflé

Naïve interpreter implementations for imperative languages may run hundreds or thousands of times slower than the equivalent compiled code [18, 44], even with various advanced implementation techniques to speed up their performance [6, 14, 20, 21]. However, the research literature has little to offer on how to design and implement a high-performance interpreter for Datalog languages such as Soufflé [28, 47]. One of the main challenges is to find an in-memory representation of relations for performing set manipulation and access operations efficiently, since the interpreter may execute such operations billions of times for a single rule.

Soufflé's synthesized C++ code can cope with such large quantities of relational operations due to specialized in-memory data structures [29–31, 40] called Datalog-Enabled Relational (DER) data structures [31]. A DER is an abstract data type for storing tuples of fixed length as a set. The abstract data type facilitates efficient inserts, membership tests, and enumerations. In particular, DER data types offer efficient range queries for predefined tuple orders. We refer to these specialized range queries as *primitive searches*, which are the most essential and performance-critical building blocks for executing Datalog rules [48]. For each relation in the Datalog program, the synthesizer specifically tailors a DER data structure for a relation using C++ template parameters. The parameters specify the DER's implementation (B-tree, trie, etc.), shape (the arity and types of the tuple elements), and tuple orders/operations. While a synthesizer can generate static code using C++ templates, an interpreter cannot directly utilize the templated data structures because of their static nature. A new approach is necessary to *de-specialize* these data structures so that a virtual execution environment can pre-compile and adopt them. Further challenges include optimizing the interpreter's instruction dispatch and overcoming the lack of optimization that a synthesizer would otherwise allow.

Thus, the key challenges addressed by our work are: (1) finding an approach for interpreters to benefit from the deeply specialized primitives available to compiled engines, (2) minimizing interpretation overheads of orchestrating primitive operations, and (3) reducing the performance gap between compiling and interpreting logic programming language engines. In this work, we introduce the Soufflé Tree Interpreter (STI), which is implemented in the style of an Abstract Syntax Tree (AST) interpreter [3]. The STI uses DER data structures that are specialized using C++ templates to overcome performance limitations of previous interpreter implementations. We also identify interpreter specific optimizations that are crucial for performance. With our new interpreter implementation techniques, the STI incurs an overhead of only $1.32 - 5.67\times$ on real-world benchmarks when compared to the synthesized C++ code. If compilation time is taken into account, the interpreter can be $6.46\times$ faster on average. This demonstrates its effectiveness for scenarios where the synthesizer may not be preferred due to the large overhead of the compilation process.

The contributions of this work are as follows:

1. We introduce an efficient tree interpreter implementation for Soufflé and explain its key design features.
2. We show how to *de-specialize* templated C++ data structures and how to use them in a virtual execution environment.
3. We provide four optimization techniques that improve the performance of our interpreter and present their impact on the overall performance.

Our de-specialization technique is not necessarily specific to C++ and Souffle, and can be applied more generally whenever there are templated data structures in language implementations with a large parameter space. With our technique, these data structures can be instantiated at runtime by reducing the parameter space and enabling the implementation of a fast interpreter.

2 Background

Datalog [1, 15] is a fragment of first-order logic with recursion. A Datalog program is a set of Horn clauses [36] of the form $L_0 :- L_1, \ldots, L_n$ where each L_i is a *literal* of the form $R_i(x_0, \ldots, x_m)$ with *relation* R_i of *arity* m. Each *argument* x_i can either be a constant or a variable. We refer to literal L_0 as the *head* and literals $L_1, ..., L_n$ as the *body* of the clause. When the body of the clause is empty, it represents a *fact*; otherwise, the clause represents a *rule*. Facts are also known as an *Extensional Database* (EDB) in Datalog and are unconditionally true [15]. For example,

$$parent(Bob, Alice).\ parent(Alice, Carol).$$

represents facts that introduce the two tuples $(Bob, Alice)$ and $(Alice, Carol)$ into relation *parent*.

The body of a rule consists of a set of literals joined by conjunctions. The interpretation of a rule is as follows: If all the literals in the body hold, then the head holds as well. The facts derived from rules are called *Intensional Database* (IDB). For example, the rule

$$grand\_parent(X, Z) :- parent(X, Y), parent(Y, Z)$$

reads that X is Z's grandparent if X is the parent of Y, and Y is the parent of Z. Such a rule is evaluated by finding facts matching the body literals, and generating the corresponding head fact.

Datalog also allows recursion and negation. A rule is recursive if the relation of the head appears in the body (i.e., the relation depends on itself). Rules may be mutually recursive if their dependencies form a cycle. Negations appear in the form of a negated literal, which holds true if the corresponding positive literal does not hold. Datalog defines the semantics of negation by stratification [1, 16, 50].

Soufflé. Soufflé [28, 47] is an extension of Datalog that includes functors (e.g., string operations, binary constraints, arithmetic operations), aggregates, components, typed structures (i.e., records), and facilitates debugging [54]. Soufflé evaluates a program in phases as shown in Fig. 1. In the first phase, Soufflé takes a Datalog program as input and generates an Abstract Syntax Tree (AST). In this stage, Soufflé checks the input program for syntax and semantic errors and applies high-level Datalog optimizations if possible. In the next phase, Soufflé translates the AST to a Relational Algebra Machine (RAM) program [47]. The RAM representation combines elements of relational algebra queries and control flow to describe the Datalog program in an imperative/relational fashion. The RAM representation ensures efficient pre-runtime optimizations, such as automatically computing indices for fast primitive searches and load-balancing for parallel computation [48]. In the last phase, the RAM program is either synthesized to a C++ program or executed directly using an interpreter.

A RAM program is represented as a tree structure containing relation declarations, subroutines, and the main program. The main program forms the root of the tree and calls subroutines to execute each part of the program. Each subroutine then contains relational operations that allow the execution of Datalog rules, such as encoding primitive searches, filters, and insertions.

Fig. 2 illustrates a simple security analysis application written in Soufflé. The Datalog code identifies potentially vulnerable regions in a program that are unprotected by a security check. From Rule 1, a code block y is unsafe if there is an edge from an unsafe code block to y and y is not protected. From Rule 2, a violation is a code block that is vulnerable and unsafe.

```
Unsafe("while").
/* Rule 1 */                /* Rule 2 */
Unsafe(y):-                 Violation(x):-
    Unsafe(x),                  Vulnerable(x),
    Edge(x, y),                 Unsafe(x).
    !Protect(y).
```

Figure 2. Datalog Example: Security Analysis

Fig. 3 shows the fragment of the RAM representation for Rule 1 of the example. In Line 1, the RAM program inserts a tuple into the input relation *delta_Unsafe*. The next step is to generate the knowledge for the *Unsafe* relation, which corresponds to Rule 1 in the Datalog program (Fig. 2). Since this rule is recursive, the rule evaluation uses a loop (Line 4). The rule evaluation begins by checking for emptiness of the body relations (Line 5). Then, a simple nested loop join iterates through all tuples in *delta_Unsafe*, and finds the matching tuples in *Edge*. If the tuple satisfies the negation for *Protect* and if it does not already exist in *Unsafe*, then it is inserted into *new_Unsafe*. If no new tuples are deduced, the loop exits (Line 10). At the end of rule evaluation, the RAM program merges the newly deduced tuples (*new_Unsafe*) into the full relation (Line 11)[1], then these new tuples forms *delta_Unsafe*, which is used to compute the next iteration.

```
1  INSERT ("while") INTO delta_Unsafe
2  ...
3
4  LOOP
5    IF ((NOT (delta_Unsafe = ∅)) AND (NOT (Edge = ∅)))
6      FOR a IN delta_Unsafe
7        FOR b IN Edge ON INDEX b.0 = a.0
8          IF ((NOT (b.1) ∈ Protect) AND (NOT (b.1) ∈
           ↪  Unsafe))
9            INSERT (b.1) INTO new_Unsafe
10   BREAK (new_Unsafe = ∅)
11   MERGE new_Unsafe INTO Unsafe
12   SWAP (delta_Unsafe, new_Unsafe)
13   CLEAR new_Unsafe
14 END LOOP
15 ...
```

Figure 3. RAM representation of the example Datalog

The primary operations in a RAM program involve querying and manipulating relations. For fast execution of RAM programs, the synthesizer produces *Datalog-Enabled Relational* (DER) data structures [31] to represent relations. The implementation, shape, and operations of the DER data structure are specialized by the synthesizer for a given input program.

[1]In the latest versions of Soufflé, the MERGE operation is replaced by a FOR loop over *new_Unsafe* whose tuples are inserted into *Unsafe*. The MERGE was eliminated to reduce the instruction set.

For example, consider the search operation on Line 7 of Fig. 3. Here, the RAM program performs a search on the *Edge* relation to find a range of tuples matching a particular value for the first element ($b.0 = a.0$). Such a filter operation constitutes a *primitive search* [48]. Therefore, the *Edge* relation should contain a data structure capable of accelerating this particular search. For example, a B-tree with a lexicographical order where the first element is first would be suitable. Such a data structure allows effective querying for a lower-bound (i.e., the first tuple where $b.0 = a.0$), and an upper-bound (i.e., the last tuple where $b.0 = a.0$). All elements between the bounds are exactly those that satisfy the condition, thus the loop can be executed in time proportional to the output size. Therefore, for this particular primitive search, the synthesizer would choose a B-tree with the lexicographical order $(a_0, a_1) \prec (b_0, b_1) \Leftrightarrow a_0 < b_0 \vee (a_0 = b_0 \wedge a_1 < b_1)$ to represent the Edge relation. While this example demonstrates the specialization of set representation and lexicographical order, the synthesizer also specializes in other aspects, such as the comparison for data types of tuple elements (int, float, etc.).

Soufflé uses C++ templates for expressing DER data structures [31]. Souffle's data structure portfolio consists of Brie [29], B-Tree [30], and equivalence relation [40]. The parameters of C++ templates instantiate the DER data structure for a concrete relation. For example, for the *Edge* relation in the above example, the synthesizer will choose a B-tree with `Comparator<0,1>` as a template to specify the lexicographical order used to store the tuples. By using templates, the synthesizer can statically optimize a relation to support the required operations. Moreover, a single lexicographical order may not be sufficient if there are multiple different search operations, so the synthesizer must generate multiple data structures for a single relation with different template instantiations. The main reason for the use of C++ templates is that parameters can be fully inlined by the C++ compiler (e.g., comparator for tuples) to obtain high performance.

3 An Interpreter for Soufflé

The Soufflé Tree Interpreter (STI) is a recursive tree interpreter that executes a RAM program. The STI employs Datalog-Enabled Relational (DER) data structures [31] that are de-specialized via adapters so that relational operations are still vastly accelerated. The STI has its own intermediate representation for a RAM program called the *Interpreter Tree*. The Interpreter Tree amends the RAM representation (stored as a tree internally) with runtime-specific information and optimizations for the interpreter. The nodes in the Interpreter Tree are called *Interpreter Nodes* (`INode`), which are lightweight nodes that directly correspond to *RAM* nodes (except in the case of super-instructions, discussed in Section 4.4), and contain execution state and pre-computed values for interpreting the RAM program. Each `INode` also encodes its

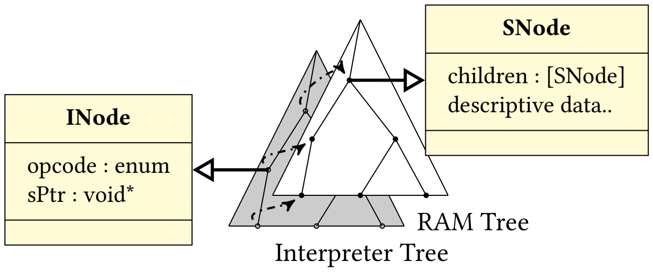


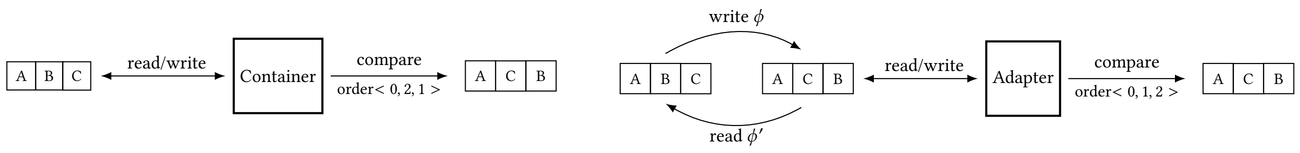
Figure 4. Interpreter tree overview, `INode` is an interpreter node, `SNode` is a source RAM node

type via an `enum` value, which allows nodes to be executed using a fast and extensible switch dispatch while enabling further optimizations (See Section 4.1). Lastly, each `INode` contains a reference, which we refer to as a *Shadow Pointer* (`sPtr`), to its corresponding *RAM* node so that the interpreter can look up static information in the RAM node during execution. The relationship between an `INode` and its RAM node is illustrated in Fig. 4. As shown in the figure, we refer to the corresponding RAM node as a *Source Node* (`SNode`). In the design of the interpreter tree, we kept the interpreter nodes lightweight, containing only what is necessary for the execution state of the interpreter.

```
1  RamDomain execute(const INode* node, Context& context) {
2      switch (node->type) {
3      case(TupleElement): {
4          const auto& ram =
5              static_cast<TupleElement>(node->sPtr);
6          size_t tupleID = ram->getTupleID();
7          size_t elementID = ram->getElementID();
8          return context[tupleID][elementID];
9      }
10     case(Max): {
11         const auto a = execute(node->leftChild())
12         const auto b = execute(node->rightChild());
13         return std::max(a, b);
14     }
15     ...
16     }
17 }
```

Figure 5. An example of executing an `INode`

Fig. 5 shows an example of executing an `INode`. Here, the `execute` method takes an `INode` and a context. The second argument, `context`, is a runtime environment that manages the creation and storage of runtime variables. Since STI supports parallel computation, the interpreter must create thread-local copies of `context`, and pass them as an argument for each invocation of `execute`. Inside the `execute` method, a switch statement is performed on the node's type to dispatch a specific node, e.g., a `TupleElement` or `Max` operation. Consider the `TupleElement` operation that accesses and returns an element from a tuple stored in a relation. The information for executing `TupleElement` is static and stored in the RAM node. In Line 4–5, a static cast (thus incurring



(a) Tuple IO in static version (Synthesizer) **(b)** Tuple IO in dynamic version (Interpreter)

Figure 6. Tuple IO actions in synthesizer and interpreter.

no runtime overhead) converts the shadow pointer from a raw pointer to the correct RAM pointer-type, and retrieves the location of the target tuple from the source node (Line 6 − 8). The other operation, Max, does not need any extra static information, and simply recursively evaluates the left and right children to return the larger result. In summary, the execution will either obtain information from the INode (i.e. execution state or pre-processed values) or static information from the RAM node via the shadow pointer.

De-specialization of Relational Data Structures. During runtime, tuples produced by the computation are stored in relations. A relation in Soufflé's generated C++ code is a collection of DER data structures [31], each of which is specialized to support the operations needed in the program. The key to the synthesizer's speed is that the DER data structures are fully specialized from a large parameter space, including the implementation (B-tree, trie, etc.), the arity, the type (integer, float, etc.) of each element, and the set of primitive searches.

The interpreter cannot adopt DER data structures directly because of their static type parameters. A solution to this problem could be to pre-generate all specialized versions for the interpreter by enumerating the parameter space. However, the combinatorially large parameter space makes a brute-force solution intractable. Therefore, we develop a framework to *de-specialize* the DER data structures and substantially reduce the parameter space. As a consequence, some of the functionality provided by the specializations must be transferred to dynamic runtime adapters, which require careful implementation so that performance is not hampered.

The first de-specialization step is the reduction of the set of all possible lexicographical orders of length N to the natural one (i.e., from 0 to $N-1$). The key observation for this reduction is that tuples can be reordered before being inserted, so that the ordering of inserted tuples determines the ordering inside the data structure, as shown in Fig. 6a and Fig. 6b. This de-specialization step also reduces the set of primitive searches. With the natural ordering, there are only N primitive searches possible: E.g., the first element is specified, the first two elements are specified, and so on. All N primitive searches can be pre-compiled for each data structure.

```
// Base class providing a virtual interface for indexes
class IndexAdapter {
public:
    virtual bool insert(Tuple) = 0;
    virtual bool contains(Tuple) = 0;
};
// An adapter class containing a BTree index
template <size_t Arity>
class BTreeIndex : public IndexAdapter {
public:
    bool insert(Tuple t) override {
        return index.insert(order.encode(t));
    }
    bool contains(Tuple t) override {
        return index.contains(order.encode(t);
    }
    ...
private:
    BTree<Arity> index;
    Order order;
};
// A factory producing indexes with arity btw. 1-16
unique_ptr<IndexAdapter> BTreeIndexFactory(size_t arity){
    switch(arity) {
        case(1): return make_unique<BTreeIndex<1>>();
        ...
        case(16): return make_unique<BTreeIndex<16>>();
        default: fatal("Size not supported yet.");
    }
}
```

Figure 7. Example of a factory and adapter

The second de-specialization step is to use only integer types in the data structures, with runtime adapters converting other types to bit representations that can fit in integers. The trade-off is that certain indexed operations can no longer be applied, since ordering for integers may not correspond to the same ordering for the target type, such as floating point and unsigned numbers.

As a result, an index can be uniquely defined by its implementation type and arity. Since there are only a handful of possible implementation types, and arities (in practice, we observed up to 16), it is now feasible to pre-compile all versions of the de-specialized DER data structures. This framework is also easy to extend in case a user needs a higher arity relation, or if a new DER data structure is introduced.

```
#define FOR_EACH(func, ...)                  \
    FOR_EACH_BTREE(func, __VA_ARGS__)       \
    FOR_EACH_BRIE(func, __VA_ARGS__)        \
    FOR_EACH_PROVENANCE(func, __VA_ARGS__)  \
    FOR_EACH_EQREL(func, __VA_ARGS__)
```

Figure 8. Macro specializing an instruction for an index type

Finally, we enable interpreter operations with those template classes by building a dynamic adapter on top of the de-specialized DER data structures which we refer to as indexes in the interpreter. Fig. 7 demonstrates the use of a factory to generate statically typed data structures, with a thin dynamic adapter around it. The base `IndexAdapter` class enforces the methods that are needed to interact with any underlying class. Then, the `BTreeIndex` implements the adapter, with the `insert` method showcasing the dynamic reordering necessary for the first de-specialization step of eliminating different lexicographical orders. The `BTreeIndexFactory` class then takes an arity, and returns a statically typed index that matches the given arity.

Another source of a potential slowdown in the de-specialized DER data structures is the iterator. Since the adapter provides virtualized behaviour, the iterator operations supported by the DER data structure must also be virtualized. However, a virtualized iterator can have huge consequences for the performance of the interpreter, since a program can easily have billions of iterator-related operations, and performing a virtual call for each operation can be very costly. To amortize the cost of the virtualization, we apply a buffer mechanism. With this mechanism, any read request on the iterator would trigger the adapter to buffer (arbitrarily chosen) 128 values from the underlying iterator, allowing the following 127 requests to operate directly on the buffer. Thus, on average, there only needs to be one virtual call to the underlying iterator for every 128 read requests.

4 Optimizations

Interpreters such as STI can perform instructions billions of times. Thus, frequently executed instructions that perform simple tasks, including allocating a heap object, dispatching a virtual function, or saving and restoring of some registers for a function call, can add significant runtime overheads. However, not all kinds of instructions are frequently executed; some are processed several orders of magnitudes more often than others. To improve the performance of an interpreter, it is important to identify frequently executed instructions and minimize their runtime (even if only few CPU cycles are eliminated by the optimizations).

One of our main techniques is to specialize instructions so that parameters of a generic instruction become constants in their specialized versions. Although these specializations seem insignificant on the surface, they can significantly reduce runtime overheads for operations on specialized data structures since the constants in the C++ code of the interpreter enable further compiler optimizations when the interpreter is translated to an executable.

Another interpreter optimization technique is replacing frequently encountered sequences of instructions with a new complex instruction that subsumes the sequence. The introduction of "super-instructions" reduces the total number of instructions executed, and reduces the costs for instruction dispatch which is a major contributing factor for the slow-down of an interpreter. In our work, we apply super-instruction techniques for a tree interpreter. In the following, we discuss the STI's optimizations in more detail.

4.1 Static Access and Instruction Generation

The adapter design enables the interpreter to interact with different types of indices during runtime uniformly. However, this strategy incurs an overhead because of the virtual interfaces and the buffer mechanism. Additionally, virtual function calls can be difficult for the compiler to inline because the target function to call is unknown until runtime. Therefore, we seek a way to eliminate the extra overhead in the adapter by enabling the interpreter to use a purely static computation model that is similar to the synthesizer.

The idea is to make the interaction between the interpreter and the data structure static, by encoding the target index's type information in the interpreter instruction set. For example, an insert operation that targets an index of type `<BTree, 3>` (B-Tree implementation with arity of 3) will have `Insert_BTree_3` as the enum value for its opcode. However, explicitly introducing all of these instructions would drastically increase the code complexity. Therefore, we use C++ macros to create specialized versions of each instruction implicitly. The central macro is shown in Fig. 8, which takes a `func` (representing a base instruction), and creates specialized versions of it for each index representation.

For each representation, additional macros create versions for each arity (in Fig. 9). Both macros also support variadic arguments, so that they can map any instruction with any arguments to specialized versions for all DER data structures that Soufflé supports.

```
#define FOR_EACH_BTREE(func, ...)\
    func(Btree, 0, __VA_ARGS__)  \
    /* ... */                    \
    func(Btree, 15, __VA_ARGS__) \
    func(Btree, 16, __VA_ARGS__)
// Equivalence relation is a specialized binary relation
#define FOR_EACH_EQREL(func, ...)\
    func(Eqrel, 2, __VA_ARGS__)
```

Figure 9. Macro specializing an instruction for each arity

For example, we can specialize the `Insert` operation into different versions as shown in Fig. 10. The `FOR_EACH` macro

```
#define GEN_ENUM(Name, Data, Arity) \
    Name##_##Data##_##Arity,
enum Token{
    FOR_EACH(Insert, GEN_ENUM)
};
```

Figure 10. Example of Enum generation

will map the GEN_ENUM Macro on all the possible combinations of arity and structures, encoding the type information of the DER structure into operation name: `Insert_BTree_0`, `Insert_BTree_1`, etc.

The final step is to generate a specialized instruction body. This is done by moving the instruction body from within the case statement into a template function. Inside each case statement, instructions are generated to call each template method. Fig. 11 shows an example, where Fig. 11a defines the call to the template method, Fig. 11b invokes FOR_EACH to generate the function calls, and Fig. 11c is the actual computation of the instruction. Note that there are several performance improvements here. Firstly, `Arity` is known at compile time; hence the array allocation at Line 4 of Fig. 11c is on the stack instead of on the heap as in the unoptimized implementation. Secondly, the loop at Line 5 can now be unrolled. Thirdly, the function call on `target` in Line 8 is not a virtual call, hence does not need virtual dispatch and can be inlined by the compiler. As a consequence, iterators no longer need a buffer mechanism, since there is no virtual call overhead anymore.

4.2 Static Tuple Reordering

As discussed in Section 3, tuples are reordered at runtime to allow the de-specialization of reducing the possible number of comparators, so that only the natural lexicographical orders are required. However, performing reordering at runtime can reduce performance. Therefore, we exploit the static nature of the RAM language. The basic idea is to rewrite the RAM representation for tuple accesses so that data is read in the natural lexicographical order of the indices. When the RAM program is produced for a Datalog program, we create an attribute env which simulates a runtime `context`. For any operation that creates a tuple t in the runtime environment, env will record the tuple and ϕ, i.e., the ordering of its elements. Later and during the lifetime of t, if another operation reads the i-th element of tuple t, the code generator will rewrite it as it is referencing the $\phi(i)$-th element, based on what env recorded. Similarly, for a query operation, env allows static reordering of the search ranges returned by the index. We do not reorder the tuple of an insertion operation statically, since a relation may consist of several indices, and we have a single insert invocation that inserts a tuple to all indices.

```
#define INSERT(Structure, Arity, ...)\
    case(I_Insert_##Arity##_##Structure):{\
        auto& rel = *static_cast\
            <Insert_##Arity##_##Structure*>\
                (node->getRelation()); \
        return evalInsert(rel, cur);\
    }
```

(a) Macro for calling template function

```
switch (node->type) {
    // Generate special instruction using FOR_EACH Marco.
    FOR_EACH(Insert)
    // Regular case that does not need specialization.
    case (Number): {
        /* .. */
    }
}
```

(b) Generate case statements in the switch loop

```
1  template<typename RelType>
2  RamDomain evalInsert(RelType& target, const SNode& cur) {
3      constexpr size_t Arity = Relation::_Arity;
4      t_tuple<RamDomain, Arity> data;
5      for (size_t i = 0; i < Arity; ++i) {
6          data[i] = eval(cur.children[i]);
7      }
8      target.insert(data);
9      return true;
10 }
```

(c) Actual function definition

Figure 11. Example of specializing an instruction

4.3 Reducing Register Pressure in Recursive Function Calls

To transfer program control, the interpreter calls the `execute` function with the child node as an argument. Since the execution model of the Soufflé interpreter is recursive, it is critical that the overhead of recursively calling `execute` is minimized. At the assembly level, once the program enters the `execute` function, it needs to allocate a new stack frame and save all the callee-saved registers on the stack to prevent instructions in the function from overwriting those registers' values. The compiler determines what registers to save by considering the instructions of the function body.

Since the `execute` function in STI contains only a long switch statement, and all of the cases can be a possible target to execute during runtime, the compiler would decide to save the maximum possible number of registers needed for the heaviest case statement. We observe that the resulting assembly code spends six instructions to save the callee-saved registers, which are not actually used by most interpreter instructions. Additionally, it spends three instructions to create a canary value to prevent stack buffer overflow attacks [13]. However, this is not always needed since many interpreter operations do not allocate a buffer on the stack.

```
#define CASE(Type) \
    case(Type): { \
        return [&]() -> RamDomain { \
#define ESAC }();}

switch (node->type) {
    CASE(Insert)
        /* Actual computation */
    ESAC(Insert)
}
```

Figure 12. Decorated case expression as a local lambda call

While these are only a few extra instructions, the interpreter often has a deeply nested recursive call structure, and these nine instructions in every recursive call can add up.

To work around the inefficiency of register allocation by the C++ compiler, we put each of the interpreter instructions inside of a local C++ lambda, to wrap it as a function call. This can be achieved by decorating the case expression using a macro as shown in Fig. 12. As a result, the compiler will note that no callee-saved registers are needed, and it will choose not to overreact. It now only stores the program stack and immediately starts execution, saving nine instructions per dispatch. The compiled code would then save registers accordingly once entering the lambda, only if required.

The trade-off here is the extra function call, but we show in the experiments that the saved instructions justify the extra function call.

4.4 Super-instructions

Super-instructions are an optimization technique to merge several small but frequent instructions into a large operation to reduce the total number of dispatches during interpretation [14, 19, 42]. The possible candidates for super-instruction optimization are combinatorially large, and different source programs can have different instruction usage, so it is critical to choose the candidates effectively in order to obtain general improvement across most programs. Our candidates for super-instruction optimizations are selected based on statistical results obtained from real-world use cases.

The key observation is that many operations, such as `IndexScan` and `Insert`, require results from runtime evaluation. Depending on the use case, the actual value being scanned or inserted can be different, but in general, they fall into three different categories:

1. `Constant`, in which case, the value to be fetched is known during the compilation of the source program and can be determined without runtime context.
2. `TupleElement`, in which case, the value is the result of some temporary tuple in the runtime environment. The actual tuple can come from user input or runtime computation, and the exact value is unknown before runtime. However, the location of target tuple in the `context` is static and thus known during compilation.
3. The last case includes all other generic expressions, e.g. arithmetic operations, for which we do not create a super-instruction, as there are many possible sub-expressions, which would introduce extra complexity into the code. Moreover, these generic expressions are comparatively rare in real-world usage.

Instead of dispatching and evaluating each sub-expression individually, a super-instruction is designed to merge `Constant` and `TupleElement` into their parent instruction to eliminate extra dispatch costs. However, a single specialized version of the operation is infeasible. For example, consider one possible parent instruction, `Insert`. An `Insert` instruction inserts a list of runtime-evaluated values into a relation as a new element; those values can come from a mixture of `Constant`, `TupleElement` and other generic expressions. A single new super-instruction cannot be used since the combination of types of underlying value expressions are not known until runtime. Therefore, generating all possible combinations of child sub-expressions for `Insert` is infeasible.

To overcome the issue, we introduce three extra fields in the interpreter node. A `constant` field that stores a list of paired elements, where the first element represents the target location to store the result, and the second element presents the actual number. Similarly, a `tupleElements` field with a list of paired elements, where the first element is the target location in the result tuple, and the second index is the location of the runtime environment where the input value should be read from. Finally, a `GenericExpression` is defined similarly. By doing this, we push the dispatch costs for `constant` and `tupleElements` from runtime to compilation time. The code generation example is illustrated in Fig. 13.

```
Node generateInsert(RamNode* node) {
    Node ret;
    for (size_t i = 0; i < num_of_operations; ++i){
        auto op = node.getChildren(i);
        if (op.type == Constant) {
            ret.addConstant((i,op));
        } else if (op.type == TupleElement) {
            ret.addTupleElement((i, op));
        } else {
            ret.addGenericExpression((i, op));
        }
    }
    /** Initialize other fields **/
    return ret;
}
```

Figure 13. Generating super-instructions for `Insert`

Our super-instructions are different to prior work [42] in the sense that we only apply it on constant and tuples, instead of all possible children of the parent operation. During runtime, we evaluate all three fields separately, as shown in Fig. 14. Because the field already implies the operation type, there is no need to perform instruction dispatch for `Constant` and `TupleElement`. The trade-off is that our approach incurs the extra cost of fetching data from additional data fields, as well as non-sequential write operations on the `result` array. However, we observe that this technique is ultimately beneficial due to the saved virtual dispatches.

```
RamDomain evalInsert(ShadowNode* node) {
    /* initialize result tuple */
    std::vector<RamDomain> tuple(n);
    // Evaluate generic expression
    for (auto& expr : node->expressions) {
        /* Rely on dispatch to evaluate result */
    }
    // Retrieve constant value
    for (auto& constant : node->constants) {
        index = constant[0];
        value = constant[1];
        tuple[index] = value;
    }
    // Retrieve tupleElement
    for (auto& t : node->tupleElements) {
        index = t[0];
        location = t[1];
        tuple[index] = getValueFromRuntime(location);
    }
    /** insert tuple into relation **/
    return result;
}
```

Figure 14. Super-instructions for an `Insert` operation

5 Performance Evaluation

To evaluate the performance of the Soufflé Tree Interpreter, we carry out several experiments. The main aim of this section is to examine the performance of the STI in comparison to the synthesizer, and also to examine the impact of each optimization technique. More specifically, we aim to answer the following research questions:

1. Can STI be considered as a fast interpreter implementation? How fast is it compared to the synthesized C++ code?
2. What contributes to the remaining gap between the performance of the interpreter and the synthesizer?
3. How much performance gain do we obtain from static instruction generation and the other optimizations?

All benchmarks are run five times on an Intel Xeon Gold 6130 CPU @ 2.10GHz. The operating system is Fedora 32, and all C++ executables are compiled with GCC 10.2.1. Throughout this section, unless specified, STI refers to the implementation with all previously discussed optimization techniques applied. For the interpreter, the execution time includes the extra code generation of the Interpreter Tree. For the synthesizer, unless specified, only the execution time of the compiled binary is measured, excluding the time for synthesizing and compiling the C++ code.

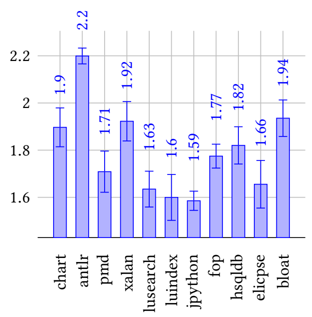
Our benchmarks are as follows:

- *Virtual Private Cloud (VPC)* benchmark [7]. A real-world network reachability reasoning tool, which provides a security analysis for Amazon's Cloud service. Soufflé is part of the logic reasoning engine in VPC.
- *DDisasm* [22] is a disassembler that produces assembly code. The disassembler engine is implemented in Datalog which Soufflé executes; it takes input a binary and reconstructs the assembly code with proper symbolic information. For the experiment, we use the *SpecCPU2006* [25] benchmarks as input data for *DDisasm.*
- *DOOP* [11] is a general and fast framework for the static points-to analysis of Java programs that produces precise analyses including context-insensitive, context-sensitive, call-site sensitive and object sensitive analyses. For this experiment, we use the *1-object-sensitive+heap* analysis along with the *DaCapo* [10] benchmark suite.

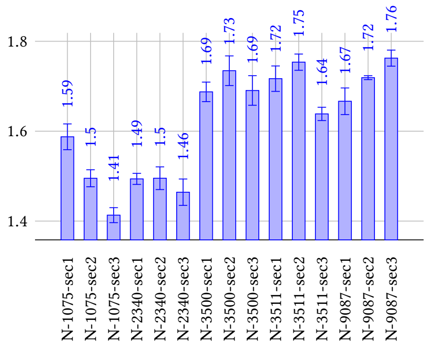
5.1 Overall Performance

The overall performance evaluation is demonstrated in Fig. 15, illustrating the relative execution time of the interpreter compared to the compiled C++ code. In DOOP and VPC benchmarks, STI execution time is 1.41× - 2.2× slower than the C++ code. The slowdown ratio in the DDisasm benchmarks ranges from 1.32× to 5.67×, except for the `specrand` benchmark, which has a slowdown up to 23×. However, `specrand` is an exceptionally short benchmark, the runtime on our machine was 0.46 seconds vs. 0.02 seconds, and the large slowdown ratio comes from the extra code generation processes (translating RAM to interpreter nodes) in the interpreter. Overall, STI is 1.32 — 5.67× slower than the compiled C++ code in the real-world use cases, demonstrating that it is an efficient interpreter and a good alternative when the synthesizer cannot be used.

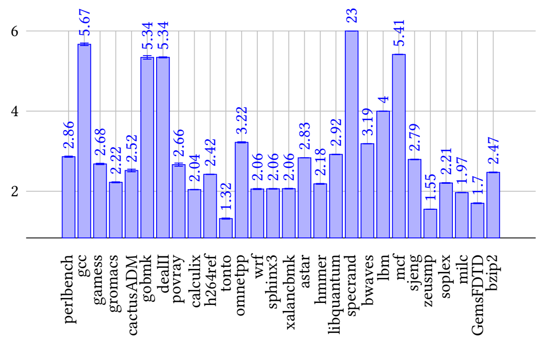
To emphasize the necessity of using static de-specialized relations in Soufflé, we also evaluate the performance of a legacy interpreter implementation. The main difference between the STI and the legacy interpreter is the comparator. The legacy interpreter uses a runtime comparator which represents the lexicographical order in an array. Due to the comparator being provided as a runtime argument, it cannot benefit from compiler optimizations. Since the comparator is executed for every internal data structure operation, performance suffers. For VPC, the legacy interpreter only finished two of the benchmarks within a 120-minute timeout, while STI finished the largest benchmark in 35 minutes. The two finishing benchmarks are 9.8× and 10.2× slower than the compiled C++ code respectively. For DOOP, the slowdown ratio of the legacy interpreter is up to 12×; and for DDisasm, the largest slowdown ratio we observed happened in gcc, which is 43× slower.



(a) DOOP slowdown ratio.



(b) VPC slowdown ratio.



(c) DDisasm slowdown ratio.

Figure 15. Execution time slowdown ratio to the synthesizer (lower is better). Y-axis is the slowdown ratio vs. the C++ runtime.

To further determine the effectiveness of STI for rapid-prototyping, we examine the time ratio between the full synthesizer runtime (compilation + execution) and interpreter runtime, i.e. the number of times the interpreter can execute a certain benchmark before the synthesizer finishes its first run. For example, a ratio of 2 means the interpreter can complete the benchmark twice before the synthesizer finishes its first computation. Table 1 summarizes the results. Most of the benchmarks in VPC has an average < 1 ratio, which means the synthesizer can likely finish before the interpreter, even when including the compilation time. Since the source Datalog program purely determines the compilation time while the execution time is determined by both the source program and the input data, benchmarks with relatively small/simple input facts tend to have a higher ratio. The main reason for a less-than-one ratio in VPC is because it contains extremely long-running benchmarks. The compilation time of the VPC source program is around 2 minutes; however, with large/complex input data, the runtime can be up to 35 minutes. For DDisasm, 90% of the benchmarks have a ratio greater than one, while the remaining benchmarks are lower than one for a similar reason to VPC — large/complex input facts diminishing the impact of compilation time. Nevertheless, with an average ratio of 15.2, it still makes the interpreter the perfect choice for prototyping in this benchmark. DOOP has an average ratio of 2.12, and all benchmarks have similar values. This is due to the Java standard library being common between all these Java projects, so the performance characteristics of points-to analysis are relatively similar between the benchmarks. All benchmarks in DOOP have a ratio larger than 1, which means for prototyping, the interpreter is always the better option. The average overall ratio, when all benchmarks are considered, is 6.46. Another factor to consider is that during the rapid prototyping and development phase, a user will typically use smaller input data, leading to a larger ratio and the interpreter becoming more effective than the synthesizer.

Table 1. Runtime ratio of each benchmark when compilation time is also considered. Higher values indicate that the interpreter is advantageous

Ratio	VPC	DDisasm	DOOP
# of values ≥ 1	20.0%	90.0%	100.0%
avg	0.79	15.2	2.12
max	1.30	42.0	2.54
min	0.62	0.44	1.63

5.2 Performance Gap

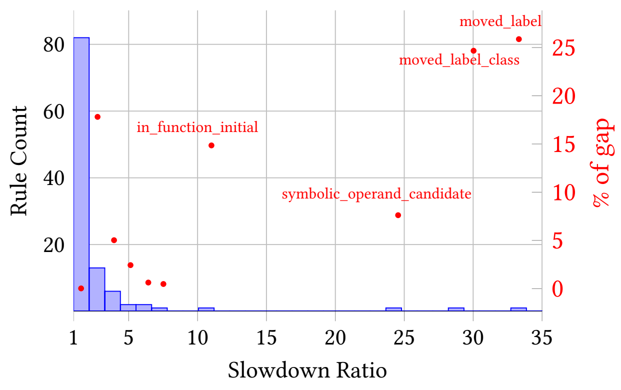


Figure 16. Bar plot is a histogram of slowdown ratio of each rule in *gamess*, with 30 bins. Points represent each bins' contribution to the total performance gap.

To further investigate the gap in performance between the interpreter and synthesizer, we perform a case study on a poorly performing benchmark: the *gamess* benchmark from DDisasm. This benchmark is chosen for demonstration purposes due to its simple profiler output, although it is not the slowest benchmark overall. The case study is carried out by using Soufflé's built-in profiler to closely investigate the computation of each rule within the Soufflé program, and to identify any performance bottleneck. We compare the slowdown of each corresponding rule in Fig. 16. The y-axis represents the number of counts of rules that are within a certain range of slowdown. The red points represent each bars' contribution to the total performance gap.

Rules that have runtime less than 0.01 second are discarded in the analysis. The majority of the rules (81) are less than 2.5× slower and in total contribute to 17.83% of the performance gap. There are 107 rules in total that have a slowdown ratio of less than 10, contributing to 27.05% of the total performance gap. The remaining bars have slowdown ratios of more than 10, each containing only a single rule, which are 10×, 23×, 29×, and 32× slower than the synthesized code respectively. These four outlier rules contribute to the majority of the performance gap - nearly 73%.

Among the outliers, the rule *moved_label* contributes the most to the performance gap, and its RAM representation is shown in Fig. 17. The main structure of *moved_label* is a loop-nest with a depth of 2 and each loop has a inner filter operation. Based on the statistics given by the Soufflé profiler, the outer for-loop is iterated 126K times and the inner-most loop is iterated 544M times. The main culprit of the slowdown here is the inner-most filter operation, with the many low-level arithmetic operations in the filter requiring 14 dispatches at the interpreter level, which are then executed 544M times, results in over 7 billion dispatches.

To confirm the hypothesis, we hand-crafted a super-instruction that computes this particular condition in the filter operation. During execution, instead of dispatching the condition, the function pointer is executed directly with the runtime environment as the function argument. As a result of the super-instructions, the number of dispatches for the filter operation drops from 14 to just 1. The performance of *moved_label* with the hand-crafted super-instructions improves significantly, reducing its computation time from 44s to only 4s. The significant improvement is due to the massive number of iterations in the inner-most loop, which amplifies the impact of only a few dispatches per loop. We observe a similar structure in the other three outlier rules as well — a condition statement that contains several dispatches, which then results in a large number of total dispatches because of a large number of inner loop iterations. For each of these outlier rules, we built custom super-instructions to reduce the dispatch overheads. Using these super-instructions, the total runtime is reduced from 331s to 209s, pushing the slowdown ratio down from 2.7 to 1.7. The same pattern is observed in other benchmarks as well, and by building super-instructions we were able to confirm the same hypothesis on them. For the slowest benchmark, *gcc*, which contains 7 such patterns, runtime is improved by 27% with custom super-instructions.

```
FOR a IN symbolic_operand ON INDEX a.3 = 131
  IF ((NOT (a.0,a.1,_,_,_) ∈
  ↪  moved_displacement_candidate)
    AND (NOT (a.0,a.1,_,_) ∈ moved_immediate_candidate))
    FOR b IN symbol ON INDEX b.0 <= a.2
      IF ((a.2 < (b.0+b.1)) AND (b.0 != a.2))
        INSERT (a.0, a.1, a.2, b.0) INTO moved_label
```

Figure 17. RAM representation of *moved_label*

5.3 Static Access and Instruction Generation

The impact of static instruction generation is shown in Fig. 18 by comparing its relative execution time against the interpreter implementation with a dynamic adapter. The improvement in performance is significant, being 24.4% faster on average and up to 55% faster, and is effective across all benchmarks. The performance improvement comes from several aspects. Firstly, the elimination of virtual function calls in the accesses to the data structure can save a considerable amount of cost, since a relation can be iterated billions of times. Secondly, better memory management is possible since the size of every runtime tuple is known at compile time, and memory can then be allocated on the stack directly instead of on the heap. Thirdly, it exposes more optimization opportunities to the compiler, since the target function to call is now static instead of dynamic.

Finally, regarding the code complexity of the technique, our approach yields only about 1800 lines of easily extensible code while the dynamic approach resulted in 2200 lines of code. The difference is mainly due to the extra complexity from the buffer mechanism and the extensive amount of sub-typing for each data structure.

5.4 Super-instructions

The performance improvement from super-instruction is shown in Fig. 19, by comparing execution time against the interpreter without super-instructions. On average, the interpreter with super-instructions achieves a 13.75% speedup. From the statistics report of the Soufflé profiler, by avoiding dispatch for evaluating constants and tuples, our super-instruction technique eliminates 22.01% of dispatches on average. The positive impact of super-instructions demonstrates the significance of dispatch overhead in the STI. In Soufflé, insertions and range queries are common operations, similar to how load operations are frequent in general-purpose languages [18, 39]. Therefore, optimizing for these common patterns by using super-instructions can make a considerable improvement.

5.5 Static Reordering and Reducing Register Pressure

Static tuple reordering brings 3.2% — 5.1% of the performance improvement based on our experiments. This improvement applies consistently across all benchmarks and further emphasizes the importance of trading off dynamic operations

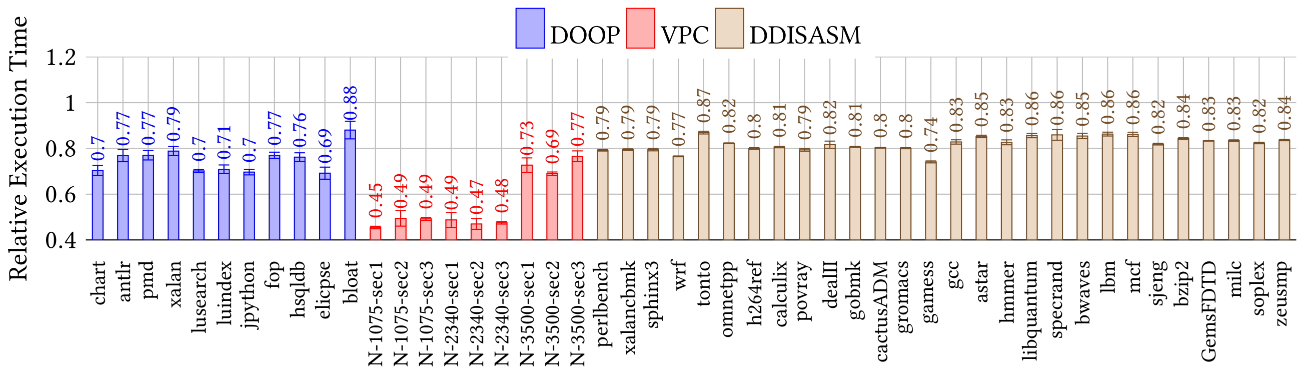


Figure 18. Performance impact of the **static interface** (lower is better). The execution time is relative to the dynamic adapter approach (runtime = 1).

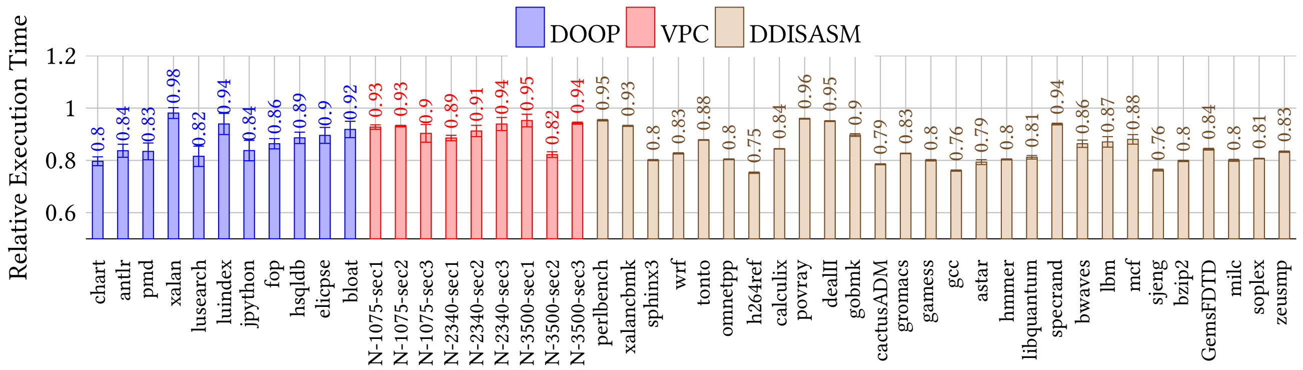


Figure 19. Performance impact of **super-instructions** (lower is better). The execution time is relative to the pre-optimized version (runtime = 1).

for static ones. The modest performance improvement is because the most frequent operation, insertion, cannot be reordered statically. An insertion operation targets a whole relation, where each underlying data structure requires a different order. One possible solution is to extend the STI instruction set to support insertion targets on a single index, which then can be reordered individually. However, this would require splitting a single relation-wise insertion into many small index-specific insertions and may significantly increase the total number of dispatches.

Meanwhile, reducing register pressure eliminates 5% - 12.5% of assembly instructions. The performance improvement is 6.3% on average, which suggests our approach of trading off extra function calls with more efficient register allocation pays off. This is because lightweight interpreter instructions which do not need any callee-saved registers are executed more frequently than heavy instructions. Overall, this optimization trick has a substantial performance impact.

6 Related Work

Datalog Engines and Logic Programming. There are many Datalog engines, including LogicBlox [5], bddbddb [35], and DDLog [45]. LogicBlox is a Datalog engine focused on business applications. While it is a rich and feature-filled language, it does not have the highly specialized data structures and optimizations that Soufflé applies in its synthesizer. Meanwhile, bddbddb translates Datalog programs into efficient binary decision diagrams (BDDs). However, manual variable ordering is key for performance in these BDDs, and automatic techniques such as Soufflé's automatic index selection [48] do not apply. On the other hand, DDLog is a compiler, that synthesizes a Differential Dataflow-based program [38] from a Datalog program, allowing for effective incremental evaluation. However, DDLog does not include an interpreter, and its compilation can often be quite slow.

Outside of Datalog, there are other logic programming languages, such as Prolog [53], XSB [46] and Flix [37]. Prolog and XSB engines are essentially compilers that produce low-level code targeted at the Warren Abstract Machine [51]. Meanwhile, Flix produces efficient code targeted at the Java

Virtual Machine. While variants of Prolog, such as SWI-Prolog [52], include an interpreter, their backward chaining approach is very different to Soufflé's forward chaining. Backward chaining relies heavily on effective unification and backtracking, while forward chaining instead requires efficient manipulation of relations.

Soufflé [28, 47] is a Datalog compiler, with its synthesizer producing high-performance parallel C++ code. The specializations and optimizations applied by Soufflé synthesizer make it the perfect platform to apply de-specializations to implement an effective Datalog interpreter.

Database Systems. The database community has been working on the efficient execution of queries using fast interpreters and JIT techniques [2, 32, 33]. However, their systems have a data-centric view with ACID properties for transactions and persistent storage for relations. In our work and others [35, 45], a Datalog program becomes a computational device that runs in isolation with volatile in-memory relations.

Interpreter Optimization. Linear bytecode representations have been shown to be effective for interpreter efficiency [18]. Hence, many performant interpreters, such as for Java [49] and Python [23] have chosen linear bytecode representations. However, the execution characteristics of Soufflé contains deeply nested loops and billions of iterations. A linear representation of Soufflé that encodes a loop with two instructions (i.e. direct jump and conditional jump) has been shown to cause high dispatch overhead [26].

Many modern interpreter optimizations, such as Python [23] and Lua [27], put focus on the dispatch loop. The general optimization approach is to either reduce the number of dispatches through super-instructions [42], or to better guide the hardware branch predictor with dispatching methods such as threaded code or indirect threaded code [9, 17]. The latter approach is becoming less effective as modern hardware is getting substantially better at branch prediction [43]. For higher abstraction level languages such as Soufflé, the dispatch cost is even less critical [12, 39]. Our experiments suggest that using indirect threaded code only brings a 3% performance improvement for Soufflé's interpreter, in the best case.

Ertl, et al. [14, 19] emphasized the impact of super-instructions on modern interpreters. They also came up with the idea of generating super-instruction dynamically during the execution, which is essentially a Just-in-Time technique (JIT). Our super-instruction technique is purely static, but the case study of the performance gap indeed suggests that STI may benefit by JITing heavily executed statements in the innermost loop.

The JIT technique [6] is also attractive because it can enable the interpreter to utilize static data structures during runtime. Nevertheless, the deployment of JIT technology for Soufflé may not be practical because of the extreme complexity of the underlying static data structures. For example, Soufflé's B-Tree has 2344 lines of templated C++ code. Another application for JIT is to deploy tailored super-instructions at runtime. For example, the hand-written super-instructions demonstrated in section 5.2 are shown to be critical for the remaining performance bottleneck in our system.

7 Conclusion

Modern Datalog engines, including Soufflé, execute logic programs efficiently because of their static optimizations for relational data structures at compile-time. However, these templated data structures cannot be employed for interpreters. This work introduces the Soufflé Tree Interpreter, which can utilize the templated Datalog Enabled Relation (DER) framework [31]. To use DER data structures in the interpreter, we de-specialize them at runtime. By reducing the possible parameter space, the DER data structures can be pre-compiled and used by the interpreter with an adapter. The idea of de-specialization is not necessarily specific to C++ and Souffle and can be applied more generally whenever there are templated data structures with a large parameter space. For example, similar techniques could be used for tensor structures in MatLab [8] for specialized template data structure in their interpreter. Another example is other relation engines such as HyPer [41] for implementing a fast interpreter.

We also identified four key optimizations that further improve the performance of the Soufflé Tree Interpreter (STI). The STI performance is evaluated using a comprehensive benchmark suite consisting of a range of real-world Datalog examples, demonstrating a slowdown of only 1.32 − 5.67 × compared to the synthesized C++ code. If the synthesizer's compile-time overheads are also considered, the interpreter can be 6.46 × faster for the first run. In the future, a promising research direction is to dynamically construct super-instructions in conjunction with light-weight JIT techniques to further reduce the performance gap between the interpreter and synthesizer.

Acknowledgments

We would like to thank Martin Bravenboer and Sam Arch for reading and commenting on a early draft of this paper. This research was supported partially by the Australian Government through the ARC Discovery Project funding scheme (DP210101984).

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (2012), 968–979. https://doi.org/10.14778/2336664.2336670

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[4] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to Soufflé: a tale of inter-engine portability for Datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, Karim Ali and Cristina Cifuentes (Eds.). ACM, 25–30. https://doi.org/10.1145/3088515.3088522

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[6] John Aycock. 2003. A Brief History of Just-in-Time. *ACM Comput. Surv.* 35, 2 (June 2003), 97–113. https://doi.org/10.1145/857076.857077

[7] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark A. Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. 2019. *Reachability Analysis for AWS-Based Networks.* Lecture Notes in Computer Science, Vol. 11562. Springer, 231–241. https://doi.org/10.1007/978-3-030-25543-5_14

[8] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231. https://doi.org/10.1137/060676489

[9] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (1973), 370–372. https://doi.org/10.1145/362248.362270

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

[12] Stefan Brunthaler. 2009. Virtual-Machine Abstraction and Optimization Techniques. *Electron. Notes Theor. Comput. Sci.* 253, 5 (2009), 3–14. https://doi.org/10.1016/j.entcs.2009.11.011

[13] Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective plus MasteringEngineering with Pearson EText – Access Card Package* (3rd ed.). Pearson.

[14] Kevin Casey, M. Anton Ertl, and David Gregg. 2007. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.* 29, 6 (2007), 37. https://doi.org/10.1145/1286821.1286828

[15] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

[16] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1990. *Logic Programming and Databases.* https://doi.org/10.1007/978-3-642-83952-8

[17] Robert B. K. Dewar. 1975. Indirect Threaded Code. *Commun. ACM* 18, 6 (1975), 330–331. https://doi.org/10.1145/360825.360849

[18] M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *J. Instr. Level Parallelism* 5 (2003). http://www.jilp.org/vol5/v5paper12.pdf

[19] M. Anton Ertl and David Gregg. 2004. Combining Stack Caching with Dynamic Superinstructions. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators* (Washington, D.C.) *(IVME '04)*. Association for Computing Machinery, New York, NY, USA, 7–14. https://doi.org/10.1145/1059579.1059583

[20] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. 2002. Vmgen - a generator of efficient virtual machine interpreters. *Softw. Pract. Exp.* 32, 3 (2002), 265–294. https://doi.org/10.1002/spe.434

[21] M. Anton Ertl and Tu Wien. 2001. Threaded Code Variations and Optimizations. In *in EuroForth.* 49–55.

[22] Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1075–1092. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[23] The Python Software Foundation. 2020. *CPython Implementation.* https://github.com/python/cpython/blob/f03d318ca42578e45405717aedd4ac26ea52aaed/Python/ceval.c#L1017

[24] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[25] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[26] Xiaowen Hu. 2020. An Efficient Interpreter for Soufflé. (07 2020). Honours Thesis.

[27] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 2007. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, Barbara G. Ryder and Brent Hailpern (Eds.). ACM, 1–26. https://doi.org/10.1145/1238844.1238846

[28] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[29] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. Brie: A Specialized Trie for Concurrent Datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2019, Washington, DC, USA, February 17, 2019*, Quan Chen, Zhiyi Huang, and Min Si (Eds.). ACM, 31–40. https://doi.org/10.1145/3303084.3309490

[30] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 327–339. https://doi.org/10.1145/3293883.3295719

[31] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2020. Specializing parallel data structures for Datalog. *Concurrency and Computation: Practice and Experience* n/a, n/a (2020), e5643. https://doi.org/10.1002/cpe.5643

[32] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[33] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *34th IEEE International Conference*

on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018. IEEE Computer Society, 197–208. https://doi.org/10.1109/ICDE.2018.00027

[34] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[35] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, Chen Li (Ed.). ACM, 1–12. https://doi.org/10.1145/1065167.1065169

[36] J. W. Lloyd. 1995. Foundations I. In *Logic Programming: The 1995 International Symposium*. The MIT Press, 177–177. https://doi.org/10.7551/mitpress/4301.003.0002

[37] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

[38] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[39] Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. 2010. Understanding the Potential of Interpreter-based Optimizations for Python. (09 2010).

[40] Patrick Nappa, David Zhao, Pavle Subotic, and Bernhard Scholz. 2019. Fast Parallel Equivalence Relations in a Datalog Compiler. In *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 82–96. https://doi.org/10.1109/PACT.2019.00015

[41] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[42] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 322–332. https://doi.org/10.1145/199448.199526

[43] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. 2015. Branch prediction and the performance of interpreters: don’t trust folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, 103–114. https://doi.org/10.1109/CGO.2015.7054191

[44] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. 1996. The Structure and Performance of Interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 150–159. https://doi.org/10.1145/237090.237175

[45] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings, Vol. 2368)*, Mario Alviano and Andreas Pieris (Eds.). CEUR-WS.org, 56–67. http://ceur-ws.org/Vol-2368/paper6.pdf

[46] Konstantinos Sagonas, Terrance Swift, and David Scott Warren. 1994. XSB as an Efficient Deductive Database Engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 442–453. https://doi.org/10.1145/191839.191927

[47] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 196–206. https://doi.org/10.1145/2892208.2892226

[48] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan D. Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (2018), 141–153. https://doi.org/10.14778/3282495.3282500

[49] Lindholm Tim, Yellin Frank, Bracha Gilad, Buckley Alex, and Smith Daniel. 2020. *The Java Virtual Machine Specification* (20 ed.). Oracle America, 500 Oracle Parkway, Redwood City, California 94065, U.S.A. An optional note.

[50] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*. Principles of computer science series, Vol. 14. Computer Science Press. https://www.worldcat.org/oclc/310956623

[51] David HD Warren. 1983. An abstract Prolog instruction set. *Technical note 309* (1983).

[52] Jan Wielemaker. 2003. An Overview of the SWI-Prolog Programming Environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments, Tata Institute of Fundamental Research, Mumbai, India, December 8, 2003 (Report, Vol. CW371)*, Frédéric Mesnard and Alexander Serebrenik (Eds.). Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Belgium), 1–16.

[53] Jan Wielemaker, Thom Fruehwirth, Leslie De Koninck, Markus Triska, and Marcus Uneson. 2012. *SWI Prolog Reference Manual (6.2.2)*. BoD – Books on Demand, USA.

[54] David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-Scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 7 (April 2020), 35 pages. https://doi.org/10.1145/3379446