Souffle With Lattices, C++ generation

Mitchel Myers and Ryan Pasculano

Department of Computer Science

CSE597: Program Analysis

Professor Gang Tan

May 7, 2021

Introduction

Souffle allows program analysis using Datalog programs. Souffle executes Datalog programs in three steps: parsing, RAM translation, and interpretation/synthesis. In the first step, a Datalog program is parsed into an Abstract Syntax Tree (AST). During RAM translation, an AST is converted into a RAM program. RAM, or Relational Algebra Machine, is an intermediate representation used by Souffle to describe how to compute the analysis described in a Datalog program. There are two options for the final step, interpretation or synthesis. For interpretation, an interpreter runs the RAM program in memory. Interpretation is notably slower, but useful for quickly testing programs. The other option is synthesis: a synthesiser generates C++ code that is equivalent to the computation described in the RAM program. Since the code is more direct than interpretation, it is much faster. Once the C++ code is compiled, it can be run on different inputs without parsing, translating, and compiling.

Lattices are a useful tool in many static analyses. However, Souffle Datalog does not have the ability to implement lattice-based analyses directly via lattices. Prior work by Qing Gong, *Extending Parallel Datalog with Lattice* [1], added the capability to declare lattices and run lattice-based analyses. However, the work only implemented the parsing, RAM translation, and interpreter for lattices. Without support for the C++ generation for lattices, the benefit of lattices is limited by the slow interpreter. The goal of this project is to complete the lattice implementation by adding lattice support into the Souffle Synthesiser.

Code

The code is made available at https://github.com/doublemix/souffle.

Methodology

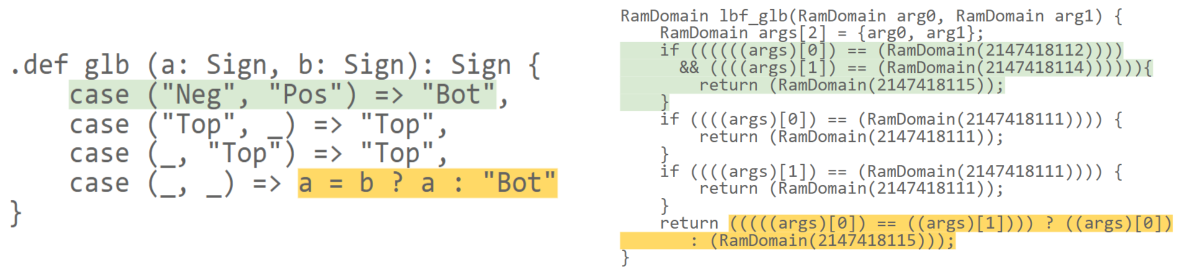
The prior work by Qing Gong defines the methodology for implementing lattices within Souffle. Syntax is added to declare enum types, which contain all possible lattice values. Syntax is also added to define lattice functions, which take either one or two lattice values and produce a new lattice value. Then, the user is able to declare a lattice by specifying its top element, bottom element, greatest lower bound function (GLB), least upper bound function (LUB). The GLB and LUB are declared as lattice functions.

For evaluation, lattice relations were built on standard Souffle relations. For a lattice relation, each possible set of non-lattice values forms a cell. The key idea is to have at most one lattice value per cell. During evaluation, multiple lattice values may enter a cell. They are reduced to a single value using the LUB after each iteration using the LATCLEAN or LATNORM operation.

To support lattices, Qing Gong added new RAM node types: RamLatticeAssociation, RamLatticeUnaryFunction, RamLatticeBinaryFunction, RamLatticeUnaryFunctor, RamLatticeBinaryFunctor, RamLatticeGLB, RamQuestionMark, RamLatClean, and RamLatNorm. For synthesis, which transforms a RAM program into C++, it was necessary to consider each of these node types.

The RamLatticeAssociation is a declarative node for storing information about the lattice within the program, such as the Top, Bottom, LUB, and GLB. As such specific code is not generated for this node, but it's data is used for generating code for other node types.

For the RamLatticeUnaryFunction and RamLatticeBinaryFunction, we generate standard C++ functions with one or two arguments. Since, the lattice functions use a case-by-case syntax, which is translated to condition-expression pairs in the RAM, we use a series of if statements to check the condition and return the corresponding value. See the figures below for an example. The RamLatticeUnaryFunctor and RamLatticeBinaryFunctor represent calls to the corresponding functions, and translate to function calls in C++. The functions are named in a collision-resistant manner. Souffle maps symbols (such as “Neg” and “Pos”) to integers, which is why the generated C++ has large integer constants in it.



```
.def glb (a: Sign, b: Sign): Sign {
    case ("Neg", "Pos") => "Bot",
    case ("Top", _) => "Top",
    case (_, "Top") => "Top",
    case (_, _) => a = b ? a : "Bot"
}
```

```
RamDomain lbf_glb(RamDomain arg0, RamDomain arg1) {
    RamDomain args[2] = {arg0, arg1};
    if ((((((args)[0]) == (RamDomain(2147418112))))
      && ((((args)[1]) == (RamDomain(2147418114)))))){
        return (RamDomain(2147418115));
    }
    if ((((args)[0]) == (RamDomain(2147418111)))) {
        return (RamDomain(2147418111));
    }
    if ((((args)[1]) == (RamDomain(2147418111)))) {
        return (RamDomain(2147418111));
    }
    return (((((args)[0]) == ((args)[1]))) ? ((args)[0])
        : (RamDomain(2147418115)));
}
```

Figure 1: Lattice function glb and generated C++ function lbf_glb

RamLatticeGLB is used when a lattice variable is bound to multiple locations in a rule body. Rather than requiring the values to be the same (as with non-lattice values), we combine the values using the GLB. For implementation, this is a call to the generated function for the GLB.

RamQuestionMark represents the ternary operator in Souffle, which was added by Qing Gong to simplify lattice function declarations. It is a straight-forward translation to the C++ ternary operator. An example of this can be seen in Figure 1.

RamLatClean and RamLatNorm are used to iterate over relations and reduce lattice values in the same cell to one value by application of the LUB. The algorithms are given in figures below. The task was to generate C++ code which was equivalent to the given algorithms. The generated code iterates the relation and calls out to the generated LUB function for reducing the values.

Algorithm 4 LATNORM algorithm

```
1: input: $R_{origin}$, output: $R_{temp}$ ($R$ is a lattice relation)
2: for each cell $S_i$ in $R_{origin}$ do
3:     apply the least upper bound function $\sqcup$ to all tuples in $S_i$ in $R_{origin}$, get $t_i$
4:     insert $t_i$ into $R_{temp}$
5: end for
```

Figure 2: The LATNORM Algorithm from [1]

Algorithm 5 LATCLEAN algorithm

```
1: input: $R_{origin}$, $R_{new}$, output: $R_{new\_temp}$ ($R$ is a lattice relation)
2: for each cell $S_i$ in $R_{new}$ do
3:     apply the least upper bound function $\sqcup$ to all tuples in $S_i$ in $R_{origin}$ and $R_{new}$, get
       the result $t_i$
4:     if $t_i \notin R_{origin}$ then
5:         insert $t_i$ into $R_{new\_temp}$
6:     end if
7: end for
```

Figure 3: The LATCLEAN Algorithm from [1]

In addition to implementing the synthesis of the nodes above a number of other supporting changes were required throughout the code to allow successful compilation of the C++ and correct output.

Evaluation

To evaluate the modifications to the Souffle synthesiser, we performed tests of constant propagation analysis and sign analysis. We used an adapted form of the program generator used by Qing Gong to create random programs. The generator produces fact file directories for programs represented by the grammar given below. The fact files are compatible with both analyses. The tests were run on an Intel i5-1035G1 1.00GHz with 4 cores and 32 GB RAM.

Constant Propagation Analysis

Programs were generated with 50, 100, 200, and 500 lines of code and ran on the compiled C++ program with 1, 2, 4, and 8 threads. The programs were also run on the interpreter for comparison to Qing Gong's work. We performed each test with 5 unique programs and calculated the average of each test.

S ::= S_1 ; S_2 | V_1 Op V_2 | V = C | IF COND: S_1 ELSE: S_2 ENDIF |
WHILE COND: S ENDWHILE | SKIP

V ::= a | b | c | d | e

Op ::= + | - | * | \

Figure 4: Program generator grammer

RESULTS

Below are the average runtimes (in seconds) by program length and thread count for constant propagation analysis.

		Compiled			
Lines	Interpreted	1 Thread	2 Threads	4 Threads	8 Threads
50	2.99	0.04	0.03	0.02	0.06
100	31.56	0.49	0.29	0.19	0.23
200	320.51	4.62	2.72	1.75	1.70
500	~3600	56.49	33.33	23.59	18.57

Table 1: Constant Propagation Results

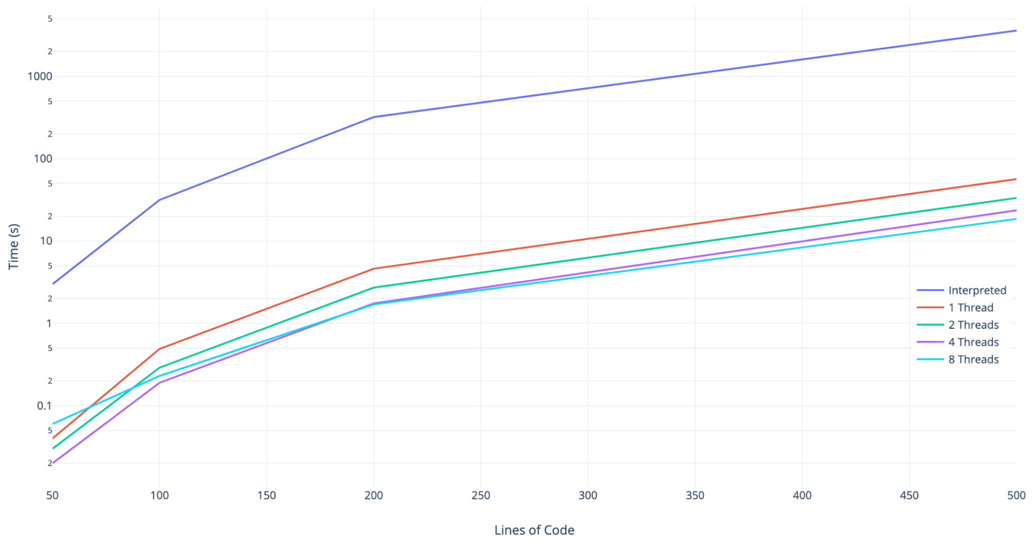


Figure 5: Interpreted and Compiled Constant Propagation Results

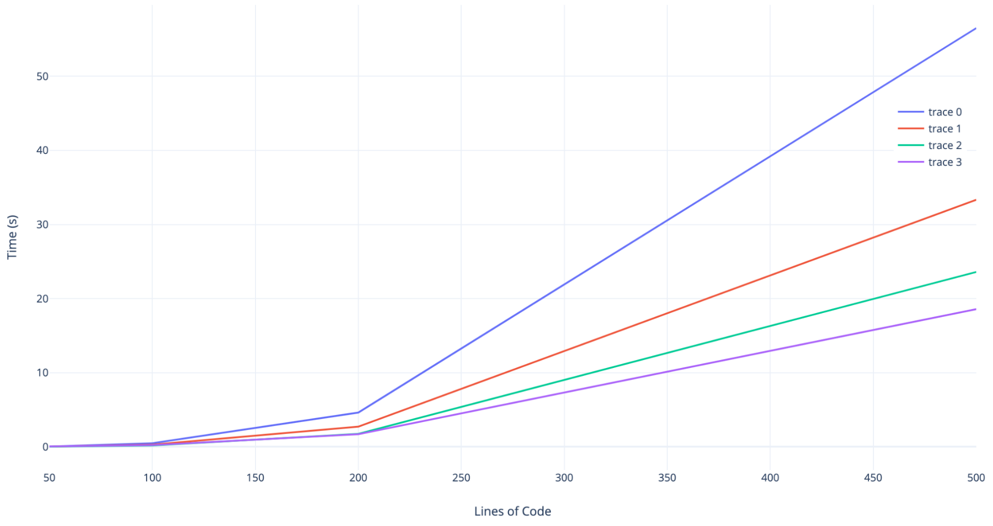


Figure 6: Compiled Constant Propagation Results

It can be seen that the compiled code significantly outperforms the interpreter by about two orders of magnitude. In each case, the increase from 1 to 2 threads reduced the execution by almost half. For shorter programs, larger numbers of threads sometimes resulted in longer executions. Since, the executions were short to begin with 1 thread, the overhead of parallelization outweighed the benefit in those cases. However, for larger programs the benefit of 4 and 8 threads can be seen. We can see that for 500 line programs we can still have reasonably good times for 1 or more threads on the compiled program, but the interpreter does not scale so well.

Sign Analysis

We also performed sign analysis using lattices. This was compared to implementation using power sets, as in the original work by Qing Gong. We ran sign analysis on the five 200 line programs with 1, 2, 4, and 8 threads for both methods.

RESULTS

Below are the average runtimes (in seconds) by method and thread count.

	Compiled			
Method	1 Thread	2 Threads	4 Threads	8 Threads
With Lattice	3.91	2.37	1.46	1.24
With Power Sets	10.84	6.40	3.98	3.27

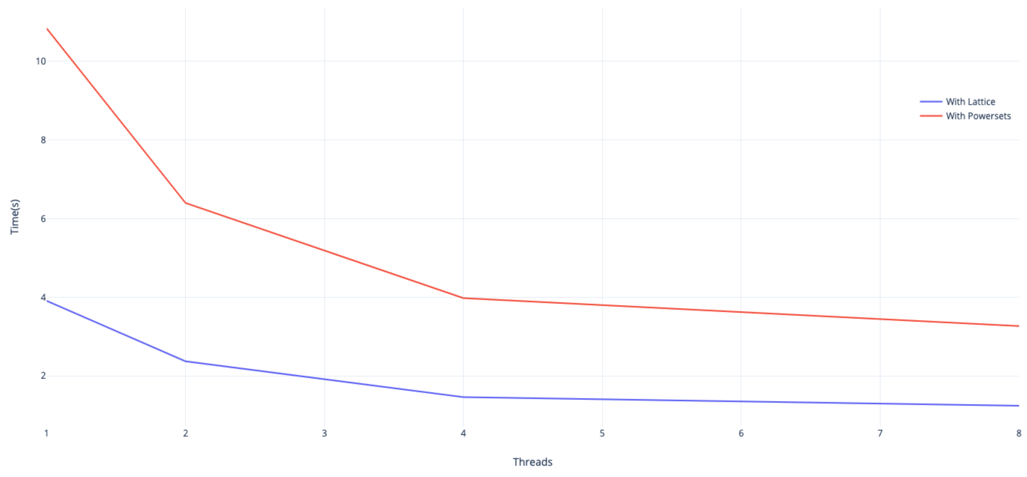


Figure 7: Sign Analysis Results

We can see from the results that use of lattices allows the analyses to run at least twice as fast, as expected, and consistent with Qing Gong's work. We also see that increasing the number of threads can increase performance, similar to constant propagation analysis.

Bibliography

[1] Gong, Q. (2020) "Extending Parallel Datalog with Lattice"