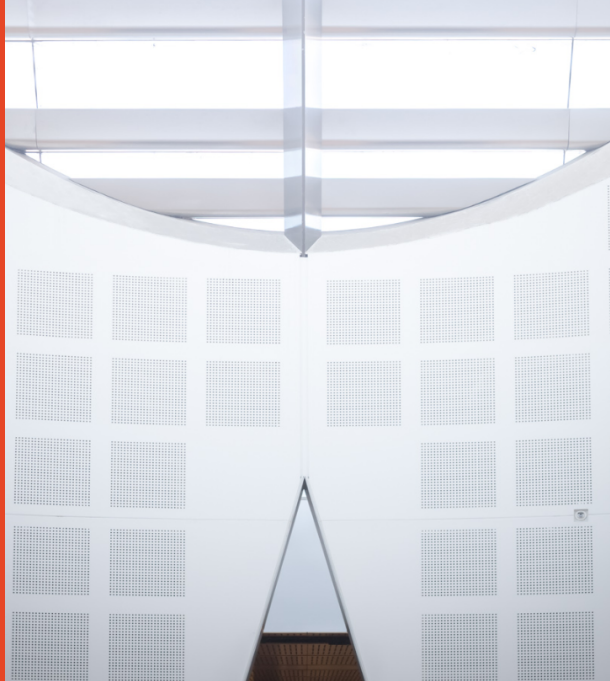
An Efficient Interpreter for Soufflé

Honours Presentation

Xiaowen Hu

July 21, 2020



THE UNIVERSITY OF SYDNEY

Introduction

1. Soufflé [8] is a logic engine.
   - Static Program Analysis [2], Security Analysis [7], Network Analysis [3].

Introduction

1. Soufflé [8] is a logic engine.
   - Static Program Analysis [2], Security Analysis [7], Network Analysis [3].
2. Soufflé's state-of-the-art synthesiser [12].
   - Translate logic programs to Relational Algebra Machine (RAM) programs.
   - Synthesise parallel C++ code from RAM.
   - High-efficient, parallel data structures [10, 9].

Introduction

1. Soufflé [8] is a logic engine.
   - Static Program Analysis [2], Security Analysis [7], Network Analysis [3].
2. Soufflé's state-of-the-art synthesiser [12].
   - Translate logic programs to Relational Algebra Machine (RAM) programs.
   - Synthesise parallel C++ code from RAM.
   - High-efficient, parallel data structures [10, 9].
3. Interpreter is necessary for Development Cycle, Debugging, and Portability
   - Synthesiser takes minutes for generating executable C++
   - Interpreter is the only option in some cloud environment
   - Current interpreter scales poorly in real-world program

Introduction

1. Soufflé [8] is a logic engine.
   - Static Program Analysis [2], Security Analysis [7], Network Analysis [3].
2. Soufflé's state-of-the-art synthesiser [12].
   - Translate logic programs to Relational Algebra Machine (RAM) programs.
   - Synthesise parallel C++ code from RAM.
   - High-efficient, parallel data structures [10, 9].
3. Interpreter is necessary for Development Cycle, Debugging, and Portability
   - Synthesiser takes minutes for generating executable C++
   - Interpreter is the only option in some cloud environment
   - Current interpreter scales poorly in real-world program

Problem

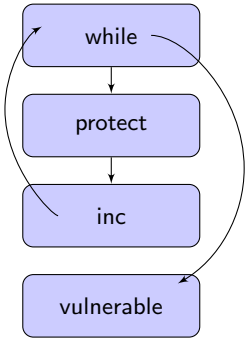
How to build interpreters that are fast and maintainable?

Logic Language

Soufflé is a variant of Datalog [1]: programmer express *what* to compute instead *how* to compute.

Security analysis in Logic

```
1  void m(int i, int j)
2  {
3    while (i < j)
4    {
5      protect();
6      ++i;
7    }
8    vulnerable();
9  }
```




```
Unsafe("while").
Unsafe(y):-
    Unsafe(x),
    Edge(x, y),
    !Protect(y).

Violation(x):-
    Vulnerable(x),
    Unsafe(x).
```

Relation Algebra Machine

- Soufflé translates logic programs to RAM (using Futamura Projections [13])
  - An **imperative** and **relational** program representation.
- Example rule for predicate `Violation`

  ```
  Violation(x):- Vulnerable(x), Unsafe(x).
  ```

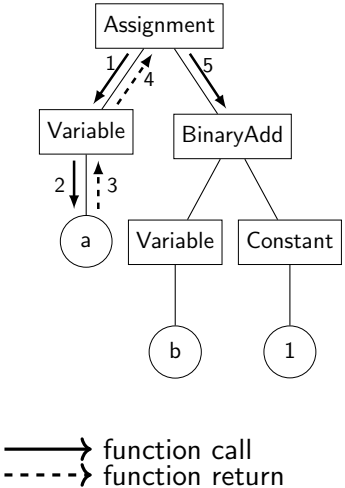
- RAM Representation of example rule

```
1  IF ((NOT (Vulnerable = ∅)) AND (NOT (Unsafe = ∅)))
2    FOR a IN Vulnerable
3      IF (a) ∈ Unsafe
4        PROJECT (a) INTO Violation
```

- Soufflé's interpreter executes the RAM program.

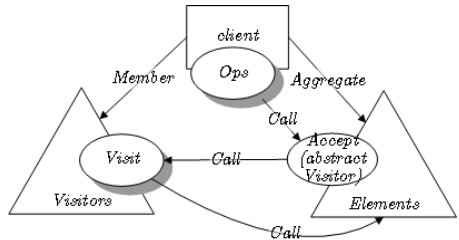
Background: AST Interpreters

- Abstract Syntax Tree (AST) Interpreter.
- Tree structure as an input.
- Recursive execution via tree traversal.



Background: AST Interpreters

- Abstract Syntax Tree (AST) Interpreter.
- Tree structure as an input.
- Recursive execution via tree traversal.
- Implemented as a Visitor Pattern in an Object-Oriented Language
  - slow because of double-dispatch

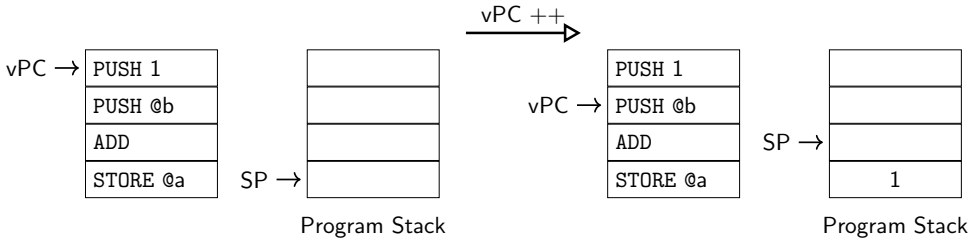


Background: Virtual Machine Interpreter

- Virtual Machine (VM) interpreter.
- Sequential code stream
  - Virtual Program Counter (vPC)
  - Increment vPC after statement execution
  - Branching by setting vPC
- Implemented with
  - Switch statement as instruction decoder
  - Variables capturing executing state including vPC

Background: Virtual Machine Interpreter

- Virtual Machine (VM) interpreter.
- Sequential code stream
  - Virtual Program Counter (vPC)
  - Increment vPC after statement execution
  - Branching by setting vPC
- Implemented with
  - Switch statement as instruction decoder
  - Variables capturing executing state including vPC



Background: Interpreter Optimisations

Many optimisation techniques are invented to improve Interpreter performance.

1. Indirect threaded code [4]
   - Utilizes `goto` statement and *label-as-value* extension.
   - Dispatch at the end of each virtual instruction - individual buffer.
   - Increase prediction rate in Branch Target Buffer (BTB).

Code Stream	Prediction result
start: A	A
B	$goto$
A	A
goto *start*	B

Figure: Indirect threaded code

Background: Interpreter Optimisations

Many optimisation techniques are invented to improve Interpreter performance.

1. Indirect threaded code [4]
   - Utilizes `goto` statement and *label-as-value* extension.
   - Dispatch at the end of each virtual instruction - individual buffer.
   - Increase prediction rate in Branch Target Buffer (BTB).
2. Super-instruction [5]
   - Build specialized instruction by amalgamating consecutive instructions.
   - Less instructions leads to less dispatch.

Code Stream	Prediction result
start: A	A
B	*goto*
A	A
goto *start*	B

Figure: Indirect threaded code

Code Stream	Prediction result
start: A_B_A	*goto*
goto *start*	A_B_A

Figure: Super-instruction

Background: Interpreter Optimisations

Many optimisation techniques are invented to improve Interpreter performance.

1. Indirect threaded code [4]
   - Utilizes `goto` statement and *label-as-value* extension.
   - Dispatch at the end of each virtual instruction - individual buffer.
   - Increase prediction rate in Branch Target Buffer (BTB).
2. Super-instruction [5]
   - Build specialized instruction by amalgamating consecutive instructions.
   - Less instructions leads to less dispatch.

Importance

As modern CPU architectures improved, branch misprediction is no longer as hurtful as a decade ago [11].

Switch-based Shadow Tree Interpreter

Traditional AST interpreter suffers several issues.

1. Shared states among multiple execution modes in AST
   - Compiler and Interpreter require different states in AST
2. Slow execution because of double dispatch in Visitor Pattern [6] (two virtual calls).

Switch-based Shadow Tree Interpreter

Traditional AST interpreter suffers several issues.

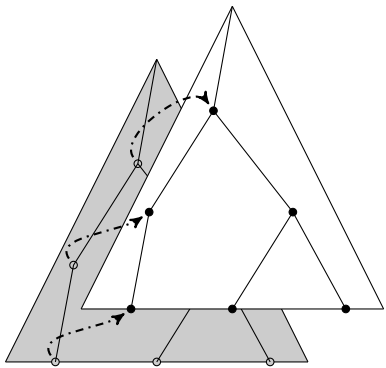
1. Shared states among multiple execution modes in AST
   - Compiler and Interpreter require different states in AST
2. Slow execution because of double dispatch in Visitor Pattern [6] (two virtual calls).

Advantages of our **Switch-based Shadow Tree Interpreter Technique**:

1. Separate descriptive tree information (i.e. AST) from execution state of interpreter with light-weight **Shadow Tree**.
2. Shadow Node contains execution state: encoding of tables / fast look-ups for interpreter
3. Switch dispatch on tagged nodes.

Switch-based Shadow Tree Interpreter

1. Shadow Tree takes the shape of Source IR (e.g. AST).
2. Each Shadow Node has a shadow pointer, referencing the source node.
3. Each Shadow Node has a Enum, representing its operation type.



Soufflé's Interpreters

- Two high-performance implementations for evaluation purposes:
  - Soufflé's Switch-based Shadow Tree Interpreter called **STI**.
  - Soufflé's Stack-based Virtual-Machine Interpreter called **SVM**.
- Require a data-structure adaptor for relational data-structure.
  - Souffle's relational data-structures are statically typed.
  - A uniform interface is required for interpreter access.
  - Unified interface highly tuned for performance.

Soufflé Tree Interpreter implementations

Soufflé Tree Interpreter (STI) utilize the SSTI strategy.

1. Shadowing the RAM.

Soufflé Tree Interpreter implementations

Soufflé Tree Interpreter (STI) utilize the SSTI strategy.

1. Shadowing the RAM.
2. Switch-dispatch on shadow nodes.

Soufflé Tree Interpreter implementations

Soufflé Tree Interpreter (STI) utilize the SSTI strategy.

1. Shadowing the RAM.
2. Switch-dispatch on shadow nodes.
3. Runtime optimisation directly implemented in Shadow Node.

Soufflé Tree Interpreter implementations

Soufflé Tree Interpreter (STI) utilize the SSTI strategy.

1. Shadowing the RAM.
2. Switch-dispatch on shadow nodes.
3. Runtime optimisation directly implemented in Shadow Node.
4. Recursive tree traversal, coarse-grained instruction set.

Soufflé Virtual Machine implementations

Soufflé Virtual Machine (SVM) utilize the stack-based Virtual Machine architecture.

1. RAM are further translated into bytecode representation.

Soufflé Virtual Machine implementations

Soufflé Virtual Machine (SVM) utilize the stack-based Virtual Machine architecture.

1. RAM are further translated into bytecode representation.
2. Switch-dispatch on bytecode.

Soufflé Virtual Machine implementations

Soufflé Virtual Machine (SVM) utilize the stack-based Virtual Machine architecture.

1. RAM are further translated into bytecode representation.
2. Switch-dispatch on bytecode.
3. Runtime information stored in separated data structure.

Soufflé Virtual Machine implementations

Soufflé Virtual Machine (SVM) utilize the stack-based Virtual Machine architecture.

1. RAM are further translated into bytecode representation.
2. Switch-dispatch on bytecode.
3. Runtime information stored in separated data structure.
4. Sequential execution model, instructions are fine-grained - small and intensive.

Shadow Tree v.s. Bytecode

Shadow Tree

- Minimum implementation effort.
  - Instruction set comes for ‘free’.
  - Single pass generation.

Bytecode

- Requires extra effort.
  - Design a separate representation.
  - Multiple passes to handle virtual branch.

Shadow Tree v.s. Bytecode

Shadow Tree

- Minimum implementation effort.
  - Instruction set comes for ‘free’.
  - Single pass generation.
- Runtime optimisations are maintainable.
  - Execution state inserted into Shadow Nodes, without violating engineering concern.
  - Optimisation does not break error reporting.
  - Complexity of the algorithm well contained in the tree node.

Bytecode

- Requires extra effort.
  - Design a separate representation.
  - Multiple passes to handle virtual branch.
- Optimisations are tedious to implement.
  - Lose original information after name encoding.
  - Relies on separated data structure - tight coupling.

Shadow Tree v.s. Bytecode

Shadow Tree

- Minimum implementation effort.
  - Instruction set comes for 'free'.
  - Single pass generation.
- Runtime optimisations are maintainable.
  - Execution state inserted into Shadow Nodes, without violating engineering concern.
  - Optimisation does not break error reporting.
  - Complexity of the algorithm well contained in the tree node.
- Coarse-grained instruction set.

Bytecode

- Requires extra effort.
  - Design a separate representation.
  - Multiple passes to handle virtual branch.
- Optimisations are tedious to implement.
  - Lose original information after name encoding.
  - Relies on separated data structure - tight coupling.
- Fine-grained instruction set.

Data Structure Adapter

- Dynamic information to initialize statically typed data structure:
  - tuple size, indexing order, and implementation type (B-Tree, Trie, etc.)
- Features:

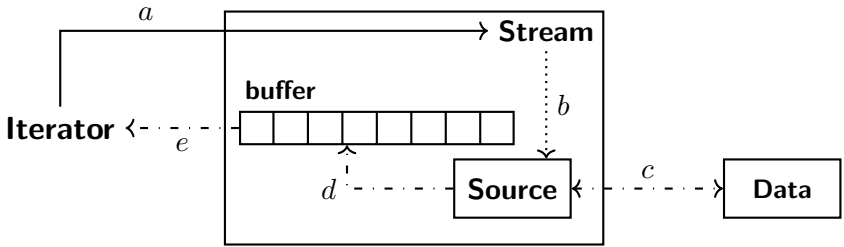
Data Structure Adapter

- Dynamic information to initialize statically typed data structure:
  - tuple size, indexing order, and implementation type (B-Tree, Trie, etc.)
- Features:
  1. Uniformed interface with a type-erased adapter.
  2. Factory method to produce data structure during runtime.
  3. Reordering tuple before reading/writing.

Data Structure Adapter

- Dynamic information to initialize statically typed data structure:
  - tuple size, indexing order, and implementation type (B-Tree, Trie, etc.)
- Features:
  1. Uniformed interface with a type-erased adapter.
  2. Factory method to produce data structure during runtime.
  3. Reordering tuple before reading/writing.
  4. Uniformed iterator with internal buffer to amortise virtual overhead.

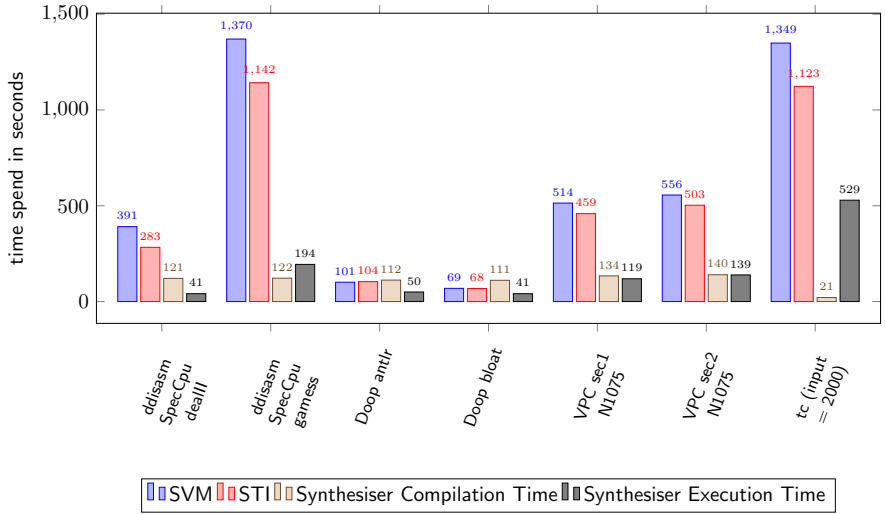
Data Structure Adapter



direct call
virtual call
data IO

a. Asking for new element.
b. Buffer is empty, trigger (virtual) read from Source.
c. Source reading data from underlying implementation.
d. Source write data into buffer.
e. Stream return data from buffer.

Performance Showdown



Impact of Instruction Set

- RAM produces heavily iterative-based computations. Under the hood, they are based on C++ iterator objects.

Impact of Instruction Set

- RAM produces heavily iterative-based computations. Under the hood, they are based on C++ iterator objects.

- In SVM, iterative statements are computed using 2 instructions: `SVM_Iterator_Read` and `SVM_Goto`.

Impact of Instruction Set

- RAM produces heavily iterative-based computations. Under the hood, they are based on C++ iterator objects.

- In SVM, iterative statements are computed using 2 instructions: `SVM_Iterator_Read` and `SVM_Goto`.

- Hence a for-loop of n iterations with one nested operation requires $3n$ dispatches.

Impact of Instruction Set (cont'd)

STI instead relies on native C++ support and recursive function calls for nested operations.

```
1 for (auto& tuple : relation) {
2     execute(node->nestedOperation);
3 }
```

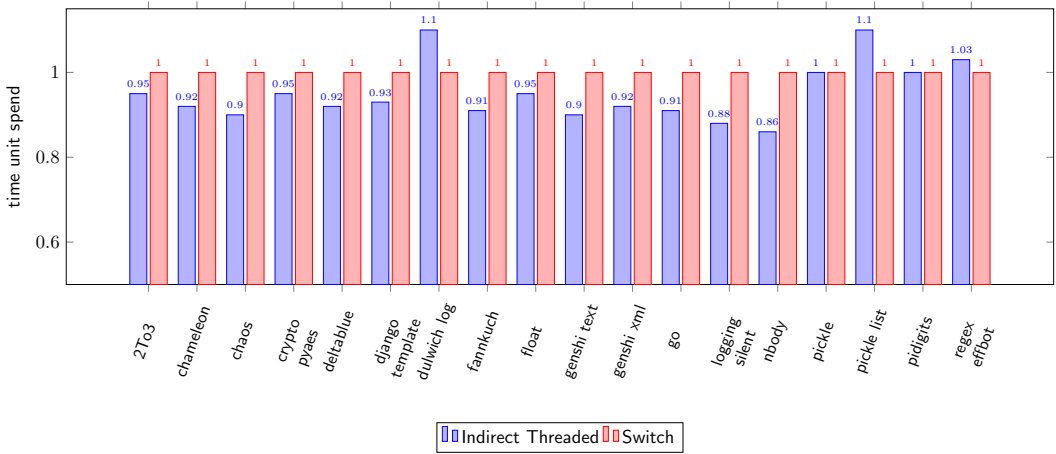
Hence a for-loop of n iterations with one nested operation requires $n + 1$ dispatches.

Experiment on Semantic Density

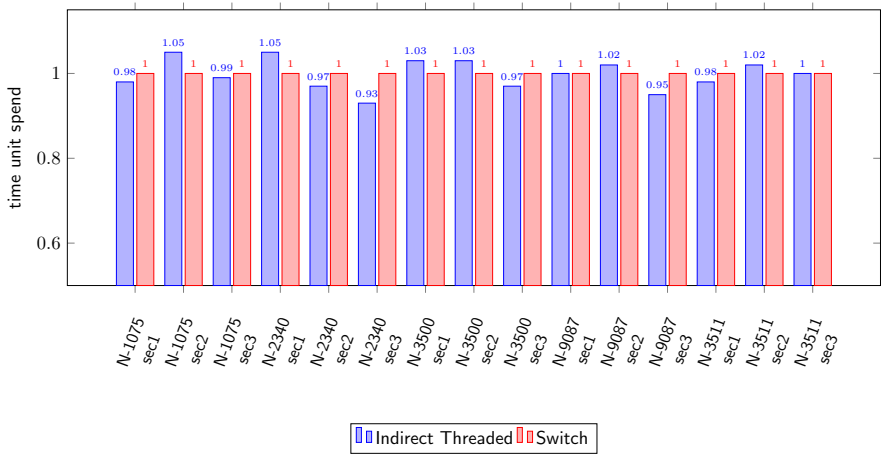
Implementation	Avg billions of dispatches per program	Avg Inst per dispatches
VPC		
SVM	19319.62	101.69
STI	17548.31	106.63
	-9.91%	+4.86%
ddisasm		
SVM	27076.065	67.05
STI	17260.909	87.89
	-36.37%	+15.56%
Doop		
SVM	633.89	109.37
STI	515.87	140.93
	-18.62%	+28.85%
tc		
SVM	164.62	87.01
STI	128.84	77.33
	-21.74%	-11.12%

Indirect Threaded Code

At the time of writting, the "Threaded Code" version is up to 15% - 20% faster then the normal "switch" version, depending on the compiler and the CPU architecture.

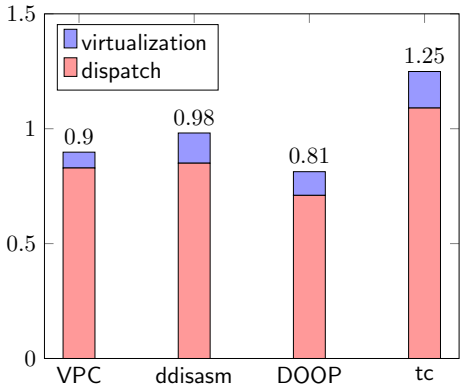


Indirect Threaded Code

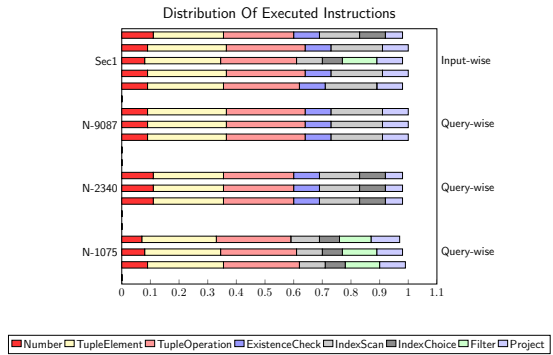


Performance Model of STI

Overhead Contributions



Operations distribution



Super-Instruction

Attribute fields to imply operation types.

```
Node generateProject(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));
        }
    }
    /** Other works **/
    return ret;
}
```

Performance result

Improvement on STI with super-instruction			
Query Program	Input Data	Reduce In Dispatch	Improvement
N-1075	sec1	26.3%	1.042
N-1075	sec2	24.1%	1.018
N-1075	sec3	26.9%	1.076
N-2340	sec1	26.2%	1.132
N-2340	sec2	26.0%	1.127
N-2340	sec3	23.7%	1.127
N-9087	sec1	27.4%	1.185
N-9087	sec2	28.5%	1.233
N-9087	sec3	26.7%	1.161
N-3500	sec1	23.5%	1.139
N-3500	sec2	22.5%	1.136
N-3500	sec3	23.5%	1.183
N-3511	sec1	26.2%	1.132
N-3511	sec2	24.3%	1.131
N-3511	sec3	21.0%	1.080

Contributions

1. A new tree interpretation strategy - Switch-based Shadow Tree Interpreter (SSTI); and a Soufflé implementation (STI).
2. A stack-based VM implementation of Soufflé (SVM).
3. A dynamically typed adaptor interface to access the statically typed data structures in Soufflé interpreter.
4. Performance evaluations of different interpreter architectures.
5. Review interpreter performance and branch optimisations on modern hardware (ITC and super-instruction).

Conclusion

1. Switch-based Shadow Tree - lightweight, efficient strategy to implement a tree-walk interpreter.
2. Soufflé Tree Interpreter is only 2.11 - 5.88 times slower then synthesiser.
3. STI is 5 - 10% faster then Soufflé Virtual Machine because of the instruction set design that fits better in the context of Soufflé.
4. Indirect threaded code has suboptimal performance on Soufflé with modern hardware.
5. Super-instruction with statistics data brings STI 10% speedup.

Reference I

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201537710.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. "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 2017. Barcelona, Spain: Association for Computing Machinery, 2017, 25–30. ISBN: 9781450350723. DOI: 10.1145/3088515.3088522. URL: https://doi.org/10.1145/3088515.3088522.

John Backes et al. "Reachability Analysis for AWS-Based Networks". In: July 2019, pp. 231–241. ISBN: 978-3-030-25542-8. DOI: 10.1007/978-3-030-25543-5_14.

Robert B. K. Dewar. "Indirect threaded code". In: *Communications of the ACM* 18.6 (1975), pp. 330–331. ISSN: 0001-0782. DOI: 10.1145/360825.360849.

M. Anton Ertl and Gregg David. "Optimizing indirect branch prediction accuracy in virtual machine interpreters". In: *ACM SIGPLAN Notices* 38.5 (2003), pp. 278–288. ISSN: 0362-1340. DOI: 10.1145/781131.781162.

Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

Reference II

Neville Grech et al. "Gigahorse: Thorough, Declarative Decompilation of Smart Contracts". In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, 1176–1186. DOI: 10.1109/ICSE.2019.00120. URL: https://doi.org/10.1109/ICSE.2019.00120.

Herbert Jordan, Bernhard Scholz, and Pavle Subotic. "Soufflé: On Synthesis of Program Analyzers". In: *CAV*. 2016.

Herbert Jordan et al. "A Specialized B-Tree for Concurrent Datalog Evaluation". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, 327–339. ISBN: 9781450362252. DOI: 10.1145/3293883.3295719. URL: https://doi.org/10.1145/3293883.3295719.

Herbert Jordan et al. "Brie: A Specialized Trie for Concurrent Datalog". In: *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM'19. Washington, DC, USA: Association for Computing Machinery, 2019, 31–40. ISBN: 9781450362900. DOI: 10.1145/3303084.3309490. URL: https://doi.org/10.1145/3303084.3309490.

Erven Rohou, Bharath Narasimha Swamy, and André Seznec. "Branch Prediction and the Performance of Interpreters -Don't Trust Folklore". In: *International Symposium on Code Generation and Optimization* (2015).

Reference III

Bernhard Scholz et al. "On Fast Large-Scale Program Analysis in Datalog". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, 196–206. ISBN: 9781450342414. DOI: 10.1145/2892208.2892226. URL: https://doi.org/10.1145/2892208.2892226.

Pavle Subotić et al. "Automatic Index Selection for Large-Scale Datalog Computation". In: *Proc. VLDB Endow.* 12.2 (Oct. 2018), 141–153. ISSN: 2150-8097. DOI: 10.14778/3282495.3282500. URL: https://doi.org/10.14778/3282495.3282500.