An Efficient Interpreter for Soufflé

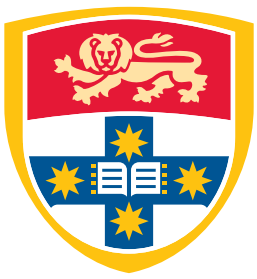
XIAOWEN HU
SID: 460048664

Supervisor: A.Prof Bernhard Scholz

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Information Technology (Honours)

School of Information Technologies
The University of Sydney
Australia

17 July 2020



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Xiaowen Hu

Signature: **Date**:

Abstract

Soufflé is a highly efficient logic engine that is used in various applications, including static program analysis, (de-)compiler tools, network analysis, and data analytics. Soufflé obtains high performance by synthesising parallel C++ from logic code. However, compilation of Soufflé can often takes several minutes, which slows down the development cycle. As a result, an efficient interpreter is desired to speed up the debugging and development process.

This work investigates various techniques to implement high-performance interpreters for Soufflé so that the performance gap between interpreted Soufflé and synthesised C++ Soufflé code can be improved. Inherently, the interpreter will always be substantially slower than the synthesised C++ code due to dispatch of instructions and virtualisation layers of data-structures. In this work, we enable high-efficient, parallel data structures in the interpreter by adding virtual adapter techniques on the statically typed data structure which could not be used during runtime in prior. We also present a new strategy for implementing a tree-walk interpreter - the *Switch-based Shadow Tree* technique. The new technique ensures a high-performance tree-walk interpreter for a coarse-grained instruction set and has sound engineering principles for the ease of development. In this work, we implement two different interpreters for Soufflé, and contrast their performance characteristics: the Soufflé's Tree Interpreter (STI) with our new Switch-based Shadow Tree technique, and a Soufflé Virtual Machine (SVM) with a standard virtual machine. STI follows better software engineering principles, easing the maintenance and future-proofs the interpreter for extensions.

We conduct experiments with real-world applications, and show that our new Switch-based Shadow Tree technique outperforms the legacy interpreter of Soufflé, which cannot complete its computation due to memory depletion and time-outs on some benchmarks. We also show that STI is only 2.11 - 5.88 times slower than the synthesised C++ code, and it is faster than SVM. We also provide a performance model that explains the gap between the synthesised C++ code and the interpreted execution of logic programs.

Acknowledgements

First of all, most enormous thank to my supervisor Bernhard Scholz. Thank you for being so patient and supportive, and thank you for helping me out, not only in academic but also in my personal life, I could not make it without you. It was such an enjoyable and pleasant experience working with you.

I also want to thank Herbert Jordan for helping me with this project, you have so many great ideas and I've learnt a lot from you; and thank you to Martin McGrane, for reviewing and refactoring my code, made me a better programmer.

I'd also like to thank Abdul Zreika, David Zhao, Ge Jin, Herbert Jordan, and Jiajun Huang for proof-reading my thesis and providing valuable opinions. Especially to Abdul and David, for offering detailed feedback and helping on the wording and grammars. You guys are so kind, and I really appreciate it.

To Ruran Jin, thank you for keeping me company and supporting me throughout those years. To my lovely friends, Xinping Miao, Huamiao Xue, Tianhao Yu, Ding Zhang, Hang Li, Qi Sheng, Zezhou Chen, Jiajun Huang, Ge Jin, Greg Ge, Dawei Tao and others, thank you for having my back and believing in me. I wish all of you all the best.

Finally, I want to thank my wonderful family. It was a very rough year, but I'm glad you were there for me.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

Soufflé (Jordan et al., 2016; Scholz et al., 2016) is a highly efficient logic engine that is used in various applications including static program analysis in Java (Antoniadis et al., 2017) and Tensor Flow programs (Sifis et al., 2020), security-oriented analysis in Ethereum Virtual Machine (Grech et al., 2019) and network analysis in Amazon Web Services (Backes et al., 2019).

Soufflé's programming language overcomes some of the limitations of classical forward chaining languages such as Datalog. For example, programmers are not restricted to finite domains, and the usage of functors and records are permitted. Soufflé obtains its high performance with its synthesiser, which compiles the source program into an equivalent, highly parallel C++ program.

However, there is an alternative mode of execution for Soufflé programs, i.e., the *Interpreter*. Unlike the compiler, which is only responsible for synthesising the source program into an equivalent and efficient form of the target language (without computing the actual result), the interpreter executes the logic program directly. In general, an interpreter breaks the execution into several steps. In the first step, a semantically equivalent, high-level, abstract intermediate representation (IR) is generated from the source program, e.g., a tree representation or a bytecode stream. The interpreter loop commences the dispatch of instructions, i.e., the interpreter is invoked, it decodes the IR one by one and executes the virtual instruction. The interpreter loop terminates when the logic program has completed the calculations. A common implementation of an interpreter is to directly execute the input program on its Abstract Syntax Tree (AST) generated by the scanner/parser.

Although the performance of an interpreter is sub-optimal compared to a highly-optimised compiled binary, it still has several advantages. Firstly, the interpreter has no compile-time costs, and can execute an input program immediately after parsing. In contrast, a compiler spends a considerable amount of time for synthesising; for example, the compilation of a Soufflé program can be substantial, exceeding multiple minutes. Secondly, being able to execute the program right away is extremely helpful in an

active project development cycle since users can obtain feedback right away instead of recompiling the source program every time when they make a change. Thirdly, no host language compiler is required for executing the input program. Once the interpreter executable is deployed on the host machine, there are no further dependencies for executing the input programs. This sometimes makes the interpreter the only option for a sophisticated system environment. For example, Amazon Web Service Lambda does not have a C++ environment; hence deploying the interpreter as an executable is the only option to have Soufflé engine running in such a cloud environment. Fourthly, interpreters are more portable and less labour-intensive to be implemented.

There has been continuous research in the implementation of efficient interpreters (Ertl and Gregg, 2001, 2003) mainly targeting imperative and object-oriented languages, but this research field has been relatively inactive in the last decade. As a consequence, most of the experiments and conclusions from this line of research is no longer valid or at least need to be revisited as modern hardware has evolved. For example, the improvement in branch-predictions units, pipelines, and cache architectures have made a significant effect on the interpreter's performance (Rohou et al., 2015).

Soufflé's logic engine also has a different style of instructions than traditional interpreters and virtual machines such as Python and Java bytecode. Soufflé's instructions set are relational algebra instructions, which can be very heavy-weight performing relational algebra operations on large tables in contrast to light-weight and small inexpensive computations in an imperative language. In addition to that, Soufflé's excellent performance depends highly on its parallel, efficient data structure which is specialised for logic evaluation (Jordan et al., 2019b, 2020). However, the data structures are statically typed classes (written in C++) and have been designed for the synthesiser. These data structures cannot be directly accessed by the interpreter during runtime due to the lack of a common interface for them. As a result, the design of Soufflé's interpreter must overcome this issue and utilise the statically typed data structure for better performance. In summary, there are different design decisions to be made, and the trade-offs differ from traditional computational engines.

In this work, we explore the design space by introducing a new tree interpretation strategy - Switch-based Shadow Tree Interpreter (SSTI). The new strategy evolved from the AST interpreter and introduces a new executable IR - the *Shadow Tree*. The shadow tree is a light-weight tree structure that takes the shape of the source IR as a tree (hence the name, shadow) and enables runtime optimisation with improved efficiency and maintainability by separating descriptive format of an AST from the execution state of an interpreter. The standard approach implementing an interpreter on AST is the use of a

visitor pattern (Gamma et al., 1994). However, using the visitor pattern causes performance issue due to the double-dispatch (i.e. two virtual calls in two class hierarchies). In addition, it implies software engineering challenges because execution state of the interpreter quite often requires modifications of the AST changing the nature of this data-structure from a descriptive format to an executable one. Our SSTI aims to solve both the performance and engineering issues.

In this work, we adapt and evolve the state-of-the-art interpreter techniques for modern CPUs and applying them in the context of Soufflé. We implement two variants of high-performance interpreters for Soufflé. Those interpreters pursue different strategies executing Soufflé programs. The first interpreter is a stack-based virtual machine interpreter whereas the second interpreter utilise the SSTI strategy.

Soufflé's Stack-based Virtual machine (SVM) commits to a linear execution order model, where programs are represented as a stream of instructions. The program computation is advanced by incrementing a virtual instruction pointer (vPC) and by conditional/unconditional goto-statements. The SVM instruction set is fine-grained, it uses small, light-weight instructions for operations, such as stack pushing/poping to pass intermediate results during computation, and a virtual jump operation to transfer control to an alternative branch.

Soufflé's Tree interpreter (STI) utilise the Switch-based Shadow Tree strategy. It applies better engineering principle compared to traditional AST interpreter and provides runtime optimisation easily by separating descriptive information from executable state. In addition to that, STI is a recursive tree interpreter where the control flow follows a recursive tree traversal pattern based on the program semantic. In contrast to SVM, the semantics of STI is more abstract, high-level and follows directly from the relational algebra machine. Instead of fully emulating the program control flow in a low-level style by utilising vPC and virtual goto statement. For example, an iterative statement is emulated directly by a dedicated tree node whereas the SVM would issue several instructions for building a loop.

Both implementations have been highly-tuned for the execution of logic programs. We show that the STI can outperform SVM in our current implementation by 7 - 15% due to its high semantic density that fits better in the relational algebra instructions. In addition to that, the tree traversal pattern in STI, which relies on light-weight tree nodes and switch dispatch, overcomes the performance overheads of the double-dispatch in the visitor pattern, and still provides great flexibility for change in the future. We conduct experiments to highlight the performance impact of these strategies.

1.1 Contributions

We summarize the contributions of this work as follows,

(1) We design and implement a stack-based Virtual Machine interpreter for Soufflé. We analyse its performance and revisit its instruction set design to hand-tailor it for a logic programming language like Soufflé. We conclude that a fine-grained instruction set may not fit well in Soufflé due to its low semantic density, which results in 10% - 36% of more dispatches compared to STI in the context of relational algebra operations.

(2) We design a new tree interpreter implementation strategy, which applies sound engineering principle. The new strategy provides great flexibility for runtime optimisations. We implement Soufflé's default interpreter based on this strategy. We conclude that a tree-walk interpreter is not necessary slower compared to Virtual Machine interpreter. In particular, the coarse-grained instruction set in STI fits well in the context of logic programming, making it 7% - 15% faster than the SVM implementation. We have deployed this interpreter into the main branch of Soufflé since it has excellent performance, and it is easy to maintain and extend, which are essential properties for Soufflé as a rapid development environment.

(3) We review optimisation techniques for interpreters, and identify the common bottleneck of interpreter performance, which is the dispatch costs. We analyze the technique not only for its impact on performance but also its implementation difficulty and maintainability. We also implement two of the techniques, namely indirect threaded code, and super-instructions (Ertl and Gregg, 2003) in Soufflé and observe their impact. We conclude that indirect threaded code is still effective in an imperative language like CPython but is sub-optimal in Soufflé. On the other hand, super-instructions, if applied carefully with support of statistics data, can result in a good performance gain.

(4) We build a performance model for STI. This model aims to provide us with a better understanding of the performance difference between interpreter and compiler. We are able to capture most of the performance gap in relatively simple benchmarks to identify the current bottlenecks (i.e.,the dispatch costs and virtualisation layer of the data structure).

(5) We build a dynamically typed adaptor classes for the statically typed data structures in Soufflé so that the interpreter can uniformly access the data. To amortize the virtualization costs of the adaptors, we design iterators with buffers and other implementation efforts to overcome the performance penalties of the adaptor.

1.1.1 Outline

The thesis is organised as follows:

In Chapter 2 we give an overview of the Soufflé language, including an introduction to Soufflé's architecture and its intermediate representation, namely the Relational Algebra Machine (RAM). In Chapter 3, we give some background about related work in interpreters, including different architectures, performance results and optimization techniques. In Chapter 4, we give the Switch-based Shadow Tree technique, we present its implementation detail and we argue about its advantages in a language like Soufflé. In Chapter 5, we give implementation detail about SVM and STI. In Chapter 6, we present out experimential results and evaluations. In Chapter 7, we conclude the work and provide potential avenues for future work.

CHAPTER 2

Soufflé

2.1 Logic Programming

Soufflé is a variant of Datalog (Abiteboul et al., 1995), which is a fragment of first-order logic with recursion and started as a Database query language. Instead of focusing on *how* to solve a problem, Datalog programmers express *what* to solve in the form of *Horn Clauses*. Horn clauses can either be facts, rules or goals. Although Datalog commenced as a Database query language, it evolved and overcame the initial restrictions such as finite domains. Its applications can be found in many fields, including program analysis (Jordan et al., 2016), Big Data (Shkapsky et al., 2016), security analysis (Whaley et al., 2005) and network analysis (Seo et al., 2013), etc.

2.1.1 Basic Structure of Logic Programs

Logic programs consists of Horn clauses of the form:

$$L_0 :- L_1, ..., L_n$$

where each L_i has the form $p_i(x_0, ..., x_m)$ for some value m; we say L_i is a *literal* with *predicts* p_i of size m, and each x_i can either be a *constant* or a *variable*. The left hand side of the clause (L_0) is referred as *head*, while the right hand side of the clause ($L_1, ...$) is referred to as the *body*. When the body of the clause is empty, it represents a *fact*; if the head is empty, it represents a goal; otherwise, the clause represents a *rule*.

Facts are also known as an Extensional Database (EDB) in Datalog, and are unconditional true. In a forward-chaining language such as Datalog, facts are usually used as the starting point of the computation:

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

In the example above, the facts mean Bob is the parent of Alice and Alice is the parent of Carol.

The body of the rule consists of a set of conjunctive predicates. A rule can be interpreted as follows: If all the predicates in the body hold, then the head holds as well. The derived facts are called Intensional Database (IDB). These facts are the computation results of some other EDB and IDB.

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

In the example above, the rule means that X is Z's grandparent if X is the parent of Y, and Y is the parent of Z.

A question mark and a dash followed by a single literal forms a goal, $?:- L.$ Goals are queries to the Database, given the rule and facts in the above paragraph and with the query:

$$?:- grand\_parent(X, Y).$$

Datalog answers all the grandparent relations in the Database - Bob is the grandparent of Carol.

In addition to that, it is also possible to have recursive rules and negation in Datalog (Abiteboul et al., 1995). Although the theory behind them can be complicated and beyond the scope of this introduction, their semantic are intuitive and should be straightforward to the reader now.

2.2 Overview of Soufflé

A logic engine like Soufflé breaks up the execution of logic programs into phases as shown in Figure 2.1. In the first phase, Soufflé takes a Datalog program as input and generates an abstract syntax tree (AST). In this stage, the input is checked for syntax and semantic errors first. After the semantic checks, a sequence of high-level optimisations is applied. In a subsequent stage, the AST IR is translated to a

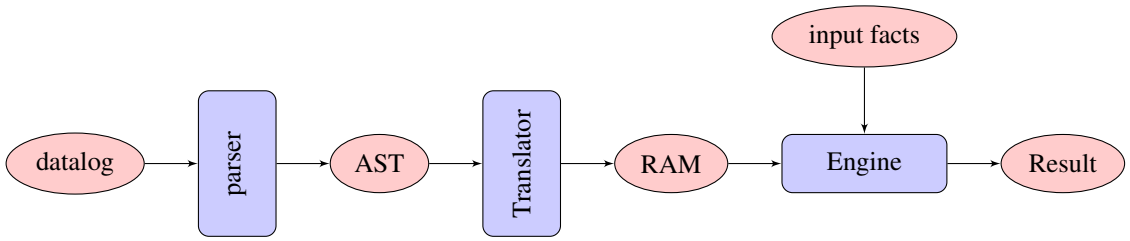
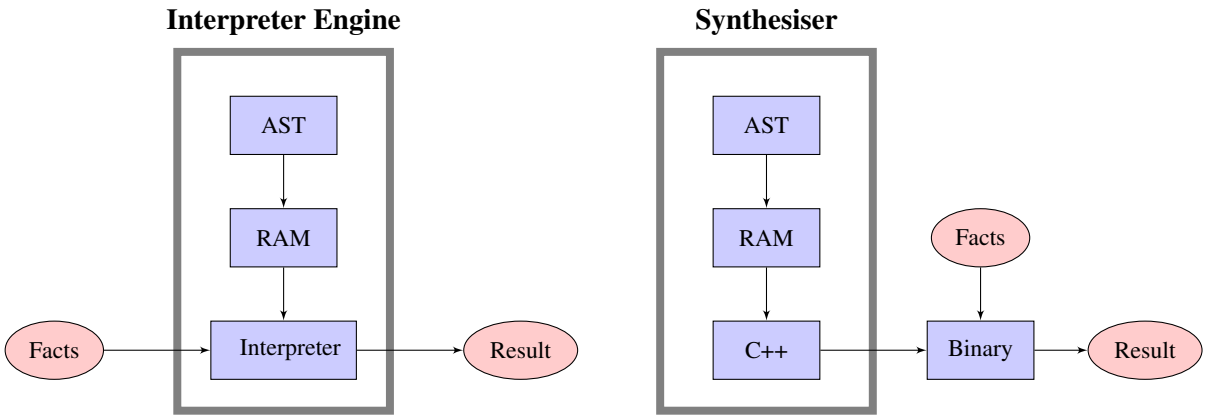


FIGURE 2.1. Soufflé execution model

Relational Algebra Machine (RAM) program, which is an imperative description of the program in the form of relational algebra queries and control flow.

In the last stage, the RAM program is either synthesised to a C++ program (Figure 2.2b), or interpreted using an interpreter as shown in Figure 2.2a.



(A) Interpreter mode (B) Synthesiser mode

FIGURE 2.2. Illustration of two engines

2.2.1 The RAM Language

Soufflè eventually translates the AST into the Relational Algebra Machine representation, which is a tree representation of the source Datalog program. RAM is designed specifically for executing relational algebra operations by an imperative programming style. The design of RAM also supports native parallel statements where a sequence of RAM statements can be executed in parallel until all the sub-statements are completed (Scholz et al., 2016). Furthermore, the static nature of RAM semantic ensures efficient pre-runtime optimisation, such as computing the index for fast range query and load-balancing in parallel computation (Subotić et al., 2018).

In detail, RAM is represented as a tree-structure in the program memory and used as a description for executing logic programs that were already lowered to an imperative form with relational algebra operations. A RAM program is consists of of relations, subroutines and the main program, where the main program is the root of the RAM tree. In the remaining part of this section, we give some typical examples of RAM language, with their semantic explanations.

RAMStatement specifies a series of actions that should be executed by RAM. Typical examples of statements in RAM are: `Sequence`, which is a compound statement consisting of a list of other statements that should be executed in order. `Sequence` also has a parallel version `Parallel`, where the child statements can be executed in parallel. In addition, `Clear` statement specify the action of clearing the content of a target relation; `Swap` statement is the action of swapping the content of two relations. Finally, `Loop` statement represents a forever loop, it is used to compute a fixed-point evaluation, along with `Exit` statement - forever computation is terminated once the exit condition is met.

RAMExpression defines the expressions in RAM. A `Number` expression returns a constant number whereas `TupleElement` accesses and returns element from a tuple access in a relation during computation. In addition to that, Soufflé enables record to support abstract user-defined compound type; a `PackRecord` is used to evaluates and store a user-define record.

RamCondition defines control flow operations in RAM. `Conjunction` defines a conjunction of conditions, evaluated to true if all the conditions are true; `Negation` simply negates the result of another *RamCondition*; `EmptinessCheck` and `ExistenceCheck` are used to check if target relation is empty, or if a tuple exists in the target relation. Finally, `Constraint` represents a binary constraint operation, such as less-equal.

The centre of the execution is the `Query` statement, representing a loop-nest of operations, which are defined by *RamOperation*. Clauses evaluation is translated into a series of nested loops in RAM. For example, consider the rule:

$$a = b \cap c$$

To derive a, RAM translates the rule into following loop-nested operations (unoptimised):

for ($x \in b$)
  for ($y \in c$)
    if ($x == y$)
      project x into a

In particular, typical loop-nested operations are: `Scan` operation iterates through all the tuples in a relation; `Choice` operation is similar to `Scan`, but stops once some condition holds; `Project` operation inserts a tuple into the target relation. Many loop-nested operations also have a `index` version to support efficient query; for example, `ScanIndex` iterates through only the tuples in the relation that follows a particular range pattern.

The base class of RAM nodes is `RamNode` and auxiliary operations are defined in order to support operations on RAM node more easily (Figure 2.3). For example, a visitor pattern is built to enables different execution models on the RAM, such as synthesising, optimisation, etc. A `clone` method is defined for cloning a source node; a `rewrite` function is built for modifying the source node and a `print` function is used for debugging and logging purpose. Finally, `getChildren` returns the children of the node and `apply` function would apply the given mapper to all the nodes.

In the following section, we provide an example resembling a security analysis and its RAM representation. We refer the reader to Appendices .1 for a detailed description about RAM semantic.

RamNode
+ getChildren() : [RamNode*] + clone() : RamNode* + rewrite(RamNode*, unique_ptr<RamNode>) : void + apply(RamNodeMapper&) : void
# print(ostream&) : void # vector<RamNode*>

FIGURE 2.3. Ram Node

2.2.2 Example

Figure 2.7 illustrates a simple security analysis application in Soufflé. The logic code identifies potentially vulnerable region in a program and will report them. On the first glance of the source code in Figure 2.4, the region may be protected because the program might execute the protect statement on

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

FIGURE 2.4. Source program

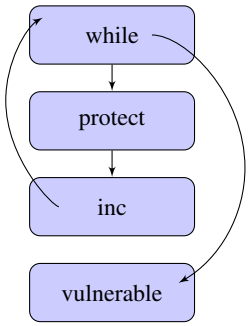


FIGURE 2.5. Control flow graph

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

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

FIGURE 2.6. Datalog program

FIGURE 2.7. Soufflé application example

line 5 before reaching the vulnerable zone. However, the control flow graph (Figure 2.5) suggests that the critical region is still accessible under an unprotected state in case of a zero-trip loop, and makes the program vulnerable; this happens when the initial condition of the while loop is not met. Although a programmer may not easily detect the security failure, it can be picked up by the Datalog program immediately.

Listing 1 shows fragments of the RAM representation of the example. The RAM representation describes the logic program in an imperative/relational format. Starting from line 1 to 3, it projects the initial data into its corresponding relations. The next step is to generate the knowledge for *Unsafe* relation, which corresponds to the logic rule of *Unsafe* in the Datalog program (Figure 2.6). In the RAM representation, a `LOOP` statement will be evaluated until the `EXIT` condition at line 9 is met (until a fixed-point is reached). Within the `LOOP` statement, there are two nested for-loops (`Scan`) that iterate through all the tuples in *Unsafe* and *Edge* relations and try to generate new knowledge about *Unsafe* based on the rule. Then, any newly generated knowledge will be inserted into *Unsafe* at line 11 - 12; RAM will keep executing the loop until no new knowledge is generated in a whole iteration. Finally, after the computation is completed, the RAM program iterates though the *Vulnerable* relation, checking if any of the tuples also exist in *Unsafe*. This corresponds to the logic rule of *violation* in the Datalog program: a statement will trigger a violation if it is a vulnerable statement and can be reached under an unsafe state. The last statement simply outputs all the knowledge about *Violation*.

Listing 1 RAM representation of the example Datalog

```
PROJECT ("while") INTO delta_Unsafe
PROJECT ("while", "protected") INTO Edge
PROJECT ("vulnerable") INTO Vulnerable
...

LOOP
 QUERY
  IF ((NOT (delta_Unsafe = ∅)) AND (NOT (Edge = ∅)))
   FOR a IN delta_Unsafe
    FOR b IN Edge ON INDEX a.0 = b.0
     IF ((NOT (b.1) ∈ Unsafe) AND (NOT (b.1) ∈ Protect))
      PROJECT (b.1) INTO @new_Unsafe
EXIT (new_Unsafe = ∅)
  QUERY
    FOR a IN new_Unsafe
      PROJECT (a.0) INTO Unsafe
    SWAP (delta_Unsafe, new_Unsafe)
    CLEAR new_Unsafe
END LOOP

IF ((NOT (Vulnerable = ∅)) AND (NOT (Unsafe = ∅)))
  FOR a IN Vulnerable
    IF (a.0) ∈ Unsafe
      PROJECT (a.0) INTO Violation

OUTPUT Violation
```

2.3 Chapter Summary

In this chapter we introduce Soufflé execution model and its underlying representation - Relational Algebra Machine. The purpose of this chapter is to give the reader a brief idea about Soufflé as a logic engine and how its semantic and execution model differs from traditional imperative language. Those characteristics eventually play an essential role in the design of the interpreter implementation, as well as the performance. In the next chapter, we present the related work and overview in the research areas of interpreter implementation.

CHAPTER 3

Background and Related Work: Interpreter and Dispatch

3.1 Overview

In this chapter, we give a detailed survey about previous work that is related to this project. To implement an efficient interpreter, it is necessary to investigate existing interpreter architectures, such as AST interpreters and the bytecode interpreters. More importantly, we would like to understand the virtual dispatch as the performance bottleneck in the interpreter - what is a virtual dispatch, and why is it slowing down the interpreter performance? Finally, we investigate the optimisation techniques that target on the interpreter architecture, instruction set design and dispatch process.

3.2 Interpreter

Interpreters are a popular technique when implementing a programming language. Compared to compilers, interpreters are easy to implement, have excellent portability and can provide active feedback in program development. However, interpreters usually run much slower than compilers (Wiedmann, 1983), so extra care must be taken regarding performance issue. While a lack-of-optimisation interpreter can be slower by a factor of 1000 than compilers, an efficient and well-optimised interpreter can be slower only by a factor of 10 (Ertl and Gregg, 2003).

An interpreter usually executes on an intermediate representation, either an Abstract Syntax Tree (AST) or a bytecode representation. The AST interpreter and bytecode interpreter differ from their implementation difficulty, semantic and performance. In the following sections, we will discuss interpreter implementations.

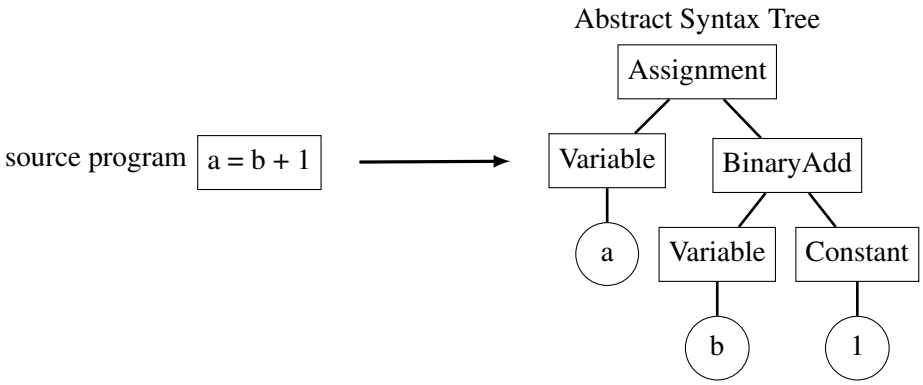


FIGURE 3.1. AST representation, some nodes are omitted for demonstration purpose

3.3 Abstract Syntax Tree Interpreter

Abstract Syntax Tree (AST) is a tree structure representation of the source program. An interpreter that directly executes on the AST representation is called an *AST interpreter*, and implementing an AST interpreter is straightforward (Aho et al., 2006; Gabbrielli and Martini, 2010). First, no effort is needed for the interpreter instruction set design as its semantics is given by AST. Second, the AST representation is 'free' because it is usually the output of a canonical language implementation, such as parser and semantic checker; Figure 3.1 gives an example of the AST generated by a parser. Finally, the software engineering technique related to AST interpreters is very mature using object-oriented language features. For example, the instruction dispatch can be done with the help of virtual function calls and the visitor pattern (Gamma et al., 1994). With this technique, an interpreter engineer will be able to produce a concise and maintainable interpreter quickly.

However, the recursive nature of an AST representation is usually considered to be unwieldy for interpreters. The tree visiting involves the recursive calls causing overheads on the program stack and runtime. In addition to that, as pointers connect tree nodes in computer memory, some would also argue that traversing AST is not cache-friendly (Ladner et al., 1999).

In the following sections, we briefly introduce two conventional approaches when implementing an AST interpreter: virtual function dispatch, which usually found on modern object-oriented languages like C++ and Java; and switch dispatch which uses tagged node and switch statement.

3.3.1 Virtual Function and Visitor Pattern

Building an AST interpreter using virtual methods is common in a language that supports polymorphic types. All tree nodes inherit from a base node type with an 'execute' interface and have its own implementations for that interface. The program is then executed by calling the 'execute' function on the root of the tree. This approach relies on the virtual dispatch of the host language: in C++ the dispatch is done by looking up through a virtual function (Bacon and Sweeney, 1996) table (vTable) in the assembly code as shown in Figure 3.2. The vTable is generated by the compiler and is hidden under the hood from the programmer. However, this also means that the programmer has no control over the dispatch mechanism. This can be a disadvantage since the execute method will be invoked frequently. Another problem wiht such a simple approach is that the interpreter's state may be deposited in the AST and the AST require the additional execute method. As a consequence, the AST becomes polluted. This is not ideal, because many other parts of the language implementation will use the AST.

To overcome the engineering issues of a simple virtual execute method in AST, a visitor design pattern was proposed as an improved strategy for implementing an interpreter. The visitor pattern is a behavioural design pattern that provides different operation implementations on a data structure without introducing new operations inside the data structure.

A visitor pattern for implementing an AST interpreter is desired as AST is usually shared among other parts of the language implementation; such as optimiser and semantic checker, hence different algorithm/implementation are needed to operate on the AST. This can be done by providing each child node with a virtual 'accept' function that takes a visitor; the visitor then chooses the appropriate 'execution' depends on the given tree node. However, the issue is that the visitor pattern introduces a double-dispatch routine: one in the 'accept' function and one in the 'execution' function, therefore, doubling the cost of dispatching during runtime.

3.3.2 Switch Dispatch

When implementing an AST interpreter in a language that does not support polymorphic type, one can always use a tag in a node and a switch statement to emulate a virtual dispatch in an object-oriented language. A tag can be expressed as an Enum variable indicating the associated AST type; a dispatch can then be performed by using a switch statement on the tag. Under the hood, the switch statement can be either lowered into a binary search or linear search on the tagged nodes, or a jump table (Berndl

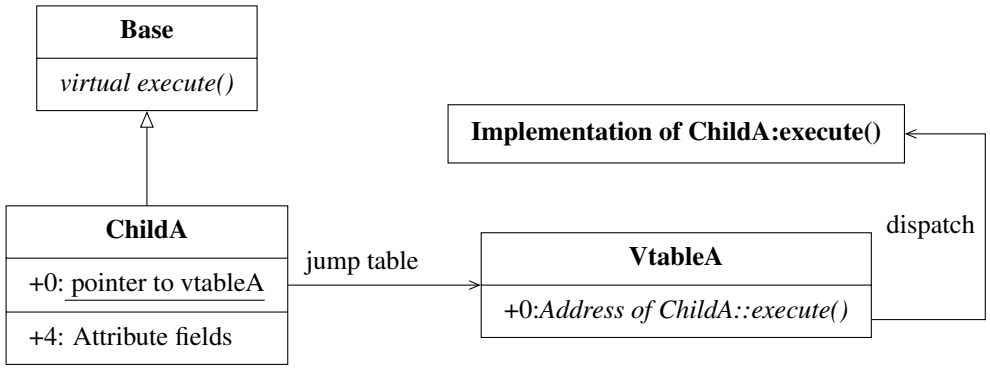


FIGURE 3.2. Example of virtual dispatch

et al., 2005; Korobeynikov, 2007). Compared to the virtual dispatch, switch dispatch does not involves function call, hence no overhead on the program stack.

In addition to that, there are multiple dispatch optimisations that can be applied on to the switch dispatch, such as Threaded Code and Selective Inlining. This makes switch dispatch a very flexible choice when implementing an interpreter. The related optimisation techniques will be discussed later in this Chapter.

3.4 Virtual Machine Interpreter

Modern computers can be complicated to manage due to their complexity. Virtual machine technology as a solution for that has been used in many fields including software development, operating system and language implementation. A virtual machine provides an interface between the hardware architecture and users, bringing flexibility and usability (Smith and Nair, 2005).

A system-level virtual machine provides the user with a completed virtualized hardware environment: they aim to emulate a physical machine. On the contrast, a process-level virtual machine is designed for a particular software, their functionality is limited, but they can be implemented in an easier manner.

A high-level language (HLL) virtual machine, as a subtype of the process-level virtual machine, is specially designed for language development. In HLL VM, the virtualization layer sits between the operating system and guest language by providing a bytecode intermediate representation, which is usually semantically close to assembly language but with a higher abstraction level. With a unified IR across different platforms, the software can be executed as long as the program can be compiled into such representation. The fundamental core of an HLL VM is usually an interpreter which directly execute

the code stream by a fetch-dispatch-execute loop. More performance-concern VMs would consider applying techniques such as JIT to execute the program on the host platform directly.

In contrast to AST, bytecode representation is a sequential representation; the computation is advanced by the control of vPC and branch statements (Torczon and Cooper, 2007). The nature of a bytecode representation is usually low-level and semantically close to assembly language. In addition to that, when designing an HLL VM, one has to decide what architecture of the virtual environment to use, either a stack-based VM or a register-based VM. The choice of the underlying architecture will not only affect how the instruction set will be designed but also the potential performance and memory consumption.

In the following sections, we briefly introduce two architectures and then discuss their difference.

3.4.1 Stack-based VM

A stack-based VM uses a virtual stack to pass intermediate result during the computation (Schoeberl, 2005). The virtual stack is usually implemented by a dynamic sized, sequential data structure, such as a list; the arguments and the intermediate results of the computation are passed between different operations though push and pop operation in a last in first out (LIFO) manner. As an example for stack-based VM and its instruction set, figure 3.3 demonstrate the code stream generated to perform a simple assignment operation, $a = b + 1$. A well-known example of stack-based VM is the Java Virtual Machine (Tim et al., 2020), which compile the source language into JVM bytecode representation and can be later executed by the interpreter.

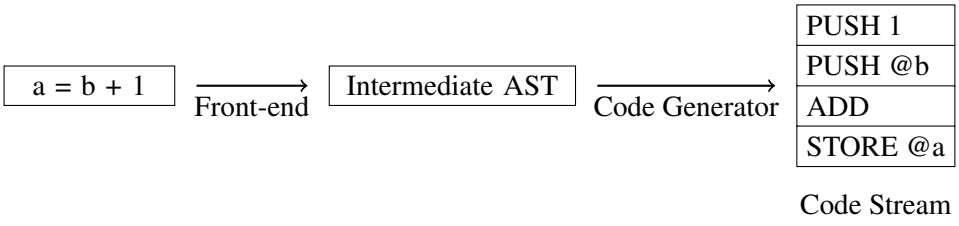


FIGURE 3.3. Example of stack VM instructions

3.4.2 Register-based VM

A register-based VM uses a sequential data structure with random accessibility to emulate the hardware machine registers; unlike real word hardware, register-based VM often has no limitation on the number

virtual registers - it can have as many as it needs (Scott, 2009). The data in register-based VM is passed between operation via 'move' and 'load' instructions, in contrast to stack-based VM, which can only access the top of the virtual program stack, there is no restriction on how/when the data can be accessed in register-based VM. The code stream generated to perform the example assignment is shown in Figure 3.4. A typical example of a register-based VM is the Dalvik Virtual Machine. In Dalvik VM, common operations usually reference up to 16 registers, in some rare case, some operation can reference up to 65535 registers (Ehringer, 2010).

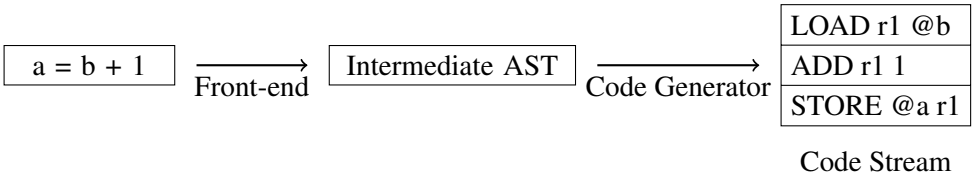


FIGURE 3.4. Example of register VM instructions

3.4.3 Stack-based VM versus Register-based VM

A big question in the design of VMs is whether a stack-based VM or a register-based VM is more efficient.

As we can see in the above examples, it requires more instructions for stack-based VM to perform the same operation as register-based. This is because stack-based memory access is implicit to the top of the stack while register-based memory access can be explicit to any register. Therefore, register-based VM can be more efficient when moving around the intermediate result during computation. However, in register-based VM, memory-related instructions tend to have larger sizes per opcode since they require extra space for arguments to specify the target registers. The difference in the size of the instruction set and opcode size plays a vital role regarding the performance.

Winterbottom et al. (Winterbottom and Pike, 1997) in their study found that compiling the source code to a register-based representation can result in efficient gain compared to a stack-based representation in Dis. Later Gregg et al. (Gregg et al., 2005) gave a more detailed experiment and quantitative data by translating a stack-based JVM to an equivalent register-based JVM. They found that the resulting register-based code reduces the total number of VM instructions by 34.88% while increasing the size of the code by 44.81%. As to extend Davis et al.'s study, Shi et al. .(Shi et al., 2008) offered a more aggressive and efficient algorithm that can translate two representations between each other. They give a comprehensive tutorial on how to do the translation using copy propagation and how to eliminate

duplicate instruction. Their experiment on JVM indicates that register-based spend 26.5% - 32.3% less time to execute the same program, with the drawback of having 25% larger bytecode size.

Despite the potential performance disadvantage, the stack-based VM still has the advantage of being easy to implement. Building an IR generator and interpreter engine based on stack-based instruction set can be very straightforward; unlike the register-based VM, which often require the input IR to be given in Static Single Assignment (SSA) form and need register allocation algorithm to help provide efficient data allocation and fetching.

To sum up, the register-based VM has a higher performance potential than stack-based VM due to its higher semantic density and the ability to access any register during runtime randomly; the performance boost comes with the cost of larger opcode size and greater implementation difficulty.

3.5 Dispatch Techniques

In the interpreter, an operation dispatch is a process of reading the next operation in the code stream and transferring control into corresponding virtual instruction for computation.

In this section, we first introduce the challenge in the interpreter dispatch process, then we examine some widely used dispatch techniques, and we discuss their characteristic in respect to their portability and efficiency.

3.5.1 Branch Prediction and Context Problem

In modern computer architecture, instruction pipelining overlap multiple instructions and execute them in parallel (Bryant and O'Hallaron, 2015). This requires the CPU to foresee the actions that will be executed in future in order to schedule the pipeline. When encounter a conditional branch, it is hard for the pipeline to schedule execution since the future operations depend on runtime result. Therefore, the branch predictor is invented as a technique to help the CPU to predict the next branch location in order to schedule the pipeline successfully. In case of a misprediction, the pipeline has to flush its content and performs a reschedule, causes a pipeline stall. In an interpreter architecture, the virtual instruction to execute depends on the code stream, whereas the code stream is generated from the source program and is only known during runtime, hence each opcode dispatch is a conditional branch in the CPU.

In 2001, Ertl and Gregg first identified that the branch misprediction as a critical performance bottleneck (Ertl and Gregg, 2001), due to highly misprediction rate during the dispatch process, which results in severe pipeline stall. They found that a bytecode interpreter usually spends 13% of the instructions on indirect branches, much higher than a regular application. Meanwhile, they also examined many available branch predictors, including profile-guided predictor, branch target buffer (BTB), BTB2 and 2-level predictors. They found none of them suited well for an application like an interpreter, where indirect jumps are very often and involve with many possible targets.

For example, the most widely used branch predictor at the time was the BTB, which contains one prediction entry for each branch location and predict the branch target by using the last jumped location. For example, for an if-statement with two possible branches, a consequence branch (*if*) is used when the condition holds, and an alternative branch (*else*) is used otherwise; if the last evaluation result leads the program to an alternative branch, then the next prediction made on the same if-statement will simply be the alternative branch again. This naive approach results in a severe misprediction rate in a program like an interpreter - imagine all operations are dispatched from a single branch location, unless two consecutive opcodes are the same, the BTB will always mispredict. Later, this particular challenge in interpreter application is named as the 'context problem' (Berndl et al., 2005)

With the discovery of the context problem, many optimizations were invented to overcome the issue. These techniques can be categorized into two kinds, one that mainly focusing on the dispatch itself and the other focus on the virtual instruction set design. We introduce some most well-known techniques in the following section.

3.5.2 Central Dispatch

Central dispatch, also known as switch dispatch if found in a standard imperative language using a switch statement, is the most natural way of writing a dispatch engine for an interpreter. The key characteristic is having a single dispatch point for dispatching all the opcode: once the computation in the virtual instruction is done, the control is given back to the central dispatcher for the next operation.

As shown in Listing 2, the dispatch point is the switch statement: it transfers program into corresponding case statement based on the opcode; after the computation , the control will go back to the top of the loop and enter the dispatch point again.

Listing 2 Example code for switch dispatch

```
#define DISPATCH() ip++; break

while (true) {

  switch(code[ip]) {
      case (ADD) : {
        // Computation goes here
        DISPATCH();
      }

      case (SUB) : {
        // Computation goes here
        DISPATCH();
      }
  }

}
```

Consider its underlying implementation in C when targeting at x86-64 using gcc 9.2, as shown in Listing 3. The dispatch is done by utilizing an address table ($.L4$) and performs the dispatch by calculating the offset of the target address in the address table (line 4 - 5). Then, the indirect jump happens at line 6. Once the virtual instruction is done, control flow is transfer into to $.L2$ and then returns to $.L9$ and ready for the next dispatch

Switch statement is available on most of the imperative language, and one can usually find an equivalent statement in other language paradigms. As long as the dispatch is always done at a single point, it can be classified as a central dispatch.

3.5.3 Threaded Dispatch

Unlike central dispatch, threaded dispatch provides a dispatch point at the end of each virtual instruction; there is no need to transfer control back to a central point, each virtual instruction can handle the dispatch by themselves.

Threaded dispatch is first described by James R. Bell (Bell, 1973) as an optimization technique to improve dispatch efficiency; it requires fewer instructions to finish a dispatch process since it does not need to transfer control back to a central dispatch point.

Listing 3 Underlying mechanism of switch statement in C

```
.L9:
        cmp     DWORD PTR [rbp-4], 4
        ja      .L9
        mov     eax, DWORD PTR [rbp-4]
        mov     rax, QWORD PTR .L4[0+rax*8]
        jmp     rax
.L4:
        .quad   .L8
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L3
.L8:
        call    A()
        jmp     .L2
.L7:
        call    B()
        jmp     .L2
.L6:
        call    C()
        jmp     .L2
.L5:
        call    D()
        jmp     .L2
.L3:
        call    D()
        nop
.L2:
        jmp     .L9
```

With the introduction of the instruction pipeline, the fewer instructions achieved in threaded dispatch became less critical to the performance. Nevertheless, later it was found to be beneficial to the indirect branch prediction in the processor (Ertl and David, 2003). At the time, the most widely used branch predictor is the branch target buffer (BTB), which uses a single buffer for each branch location and naively predict the next jump location to be the previous target of that branch location.

The central dispatch, having only a single branch location, becomes inferior here. The prediction made by BTB in central dispatch is only correct when two consecutive operations are identical in the program, which rarely happens. On the other hand, by having an individual branch location at the end of each virtual instruction, the accuracy rate of BTB can be significantly improved. Table 3.1 illustrates an example; the first column is the code stream, the prediction made by BTB before dispatching to the corresponding opcode is on the same row. The BTB in a switch dispatch fails to predict any of the

Listing 4 Example code for threaded dispatch

```
#define DISPATCH() goto **codeStream[ip++]

// codeStream = [&&ADD, ... ];

DISPATCH();

ADD:
// Computation goes here
DISPATCH();

SUB:
// Computation goes here
DISPATCH();

HALT:
// Terminate
```

Code Stream	Switch dispatch with single BTB entry.	Threaded dispatch with individual BTB entry.
start: A	goto	A
B	A	goto
A	B	A
goto *start*	A	B

TABLE 3.1. Prediction result of different dispatch methods with BTB

targets correctly as no two consecutive instructions are the same. However, in a threaded dispatch where each operation has its own BTB entry, the prediction accuracy increases dramatically. For example, since *goto* and *B* both have their own BTB entries and they are always followed by *A*, the predictions for them are always correct in each iteration.

However, implementing threaded dispatch in a modern high-level language can be troublesome. The most standard way of jumping to a dynamic program address in an imperative language is by using function pointers or a similar mechanism. The downside of this method is that it introduces new instructions on program stack when jumping to/returning from the function call. Another standard solution in C/C++ is to use goto statement and label-as-value extension in GNU C (Listing 4), but this brings a new portability issue when target machine only has ANSI C.

3.5.4 Indirect Threaded Dispatch

Dewar introduced indirect threaded dispatch or indirect threaded code (ITC) as a more memory-efficient version of threaded dispatch (Dewar, 1975). Instead of having the code stream as an array of program address (8 bytes), it creates an address table and having integer offsets (4 bytes) as an array representing their position in the address table as shown in Listing 5. When targeting on a 64-bit machine in C, this typically means we can save about 4 bytes for each opcode depends on the design of the instruction set.

Listing 5 Example code for indirect threaded dispatch.

```
#define DISPATCH() goto **addressTable[code[ip++]]

// addressTable = [&&ADD, &&SUB, &&HALT];
// code = [0, 1, ...];

DISPATCH();

ADD:
// Computation goes here
DISPATCH();

SUB:
// Computation goes here
DISPATCH();

HALT:
// Terminate
```

Standard implementation is to use the goto statement and label-as-value extension in GNU C. Although it still has a portability issue, a surprising advantage of ITC regarding software engineering concern is that, it provides an easy way of switching between central dispatch and indirect threaded dispatch when target machine does not support the latter. Since Enum type used in switch dispatch can be treated as an integer type and therefore can be directly used as the offset in ITC; switching between ITC and switch dispatch can be done by using a few lines of macro as shown in Listing 6. In fact, many modern interpreters such as CPython relies on this technique to maintain probability while aiming for better performance. In contrast to direct threaded code, when the targeted machine does not support the GNU extension, it usually requires the interpreter author with a more sophisticate method to fall back to central dispatch, since the code stream required by two dispatch methods has fundamentally different types in language like C (Pointer v.s. Integer).

Listing 6 Switching between switch / threaded dispatch

```
#ifdef LABEL_AS_VALUE
#include addressTable.h
#define DISPATCH() goto **addressTable[code[ip++]]
#define OPEATION(op) op : LABEL_##op
#else
#define DISPATCH() ip++; break
#define OPERATION(op) op
#endif

while (true) {
  switch(opCode) {
      case OPERATION(ADD) : {
        // Computation goes here
        DISPATCH();
      }

      case OPERATION(SUB) : {
        // Computation goes here
        DISPATCH();
      }
  }
}
```

3.5.5 Subroutine Threaded

Subroutine threaded was first described by Curely (Curley, 1993), who presented a Forth implementation on 68000CPU. In general, the idea of subroutine threaded is to 'glue' the instruction body with assembly code 'call' and 'return'; instead of dispatching based on a dynamic program address, code stream is loaded as a sequence of native function calls and directly to the corresponding instruction.

Subroutine threaded can reduce most of the indirect jump during the dispatches (Berndl et al., 2005), except for those relies on runtime result, such as virtual branches instruction. Although subroutine threaded involves machine dependent code, which can bring up a portability issue, the complexity of the code is well contained since it only involves porting native call and return instruction. Therefore, adding support for other platform is still relatively easy compared to porting a compiler.

3.6 Instruction Optimization

So far the optimizations we have covered were all focusing on the dispatch loop itself; another approach would be to directly alter the instruction design in order to reduce number of dispatches.

Code Stream	Prediction result	Code Stream with Superinstruction	Prediction result
start: A	goto	*start*: A_B_C	goto
B	A		
C	B		
goto *start*	C	goto *start*	A_B_C

TABLE 3.2. Superinstruction with switch dispatch and BTB

3.6.1 Super-instruction

Intuitively, the less instruction we have in the program, the less branch indirection we could encounter and hence results in fewer misprediction penalty.

Ertl and Gregg (Ertl and David, 2003) came up with the idea of super-instruction, which reduce the number of dispatches needed for the same program by building specialized super-instruction that is made up of many small instructions. As an example shown in Table 3.2, if we have a large instruction that semantically equivalent to execute A, B and C in the order, we can reduce the number mispredictions from four to two for each iteration of the loop.

If one wants to build super-instruction statically, this typically would require the author to identify frequent sequences in the program, usually with a profile-guarded approach. Nevertheless, building super-instruction dynamically is similar to applying virtual instruction threaded code; it provides more opportunities for combing basic instructions but introduces a probability issue as it involves embedded assembly code. Finally, Ertl and Gregg also found that abusing the super-instruction would result in a growth in the opcode size and can be harmful on 32-bit machines.

3.6.2 Selective Inlining

Selective inlining presented by Piumarta and Riccardi (Piumarta and Riccardi, 1998) is the most aggressive optimization technique that targets on reducing dispatch number. The idea is to translate the whole virtual program into one single instruction by concatenating each individual virtual body together. This does not require the author to generate assembly code for any interpreter instruction; it is instead done by 'memcpy'ing the machine code of each instruction body during runtime. The exact address to be copied can be determined at runtime by using GNU's label-as-value extension.

However, to keep the program counter (PC) correct, selective inlining can not deal with any relative jump to a function call, this means any instruction body with function call can not be inlined without

carefully inspect the source code. Since hardware architecture and compiler tend not to enforce any standard on whether a relative or absolute jump should be preferred, this makes selective inlining much more complicated; its portability does not depend on a particular standard which interpreter author can reference, but depends on a specific compiler implementation of a particular version. However, selective inlining can still be reliable for a virtual instruction set with fine-grained opcodes, where implementation is straightforward and does not involve many system calls.

3.6.3 Replication

Ertl and Gregg introduced replication along with super-instruction (Ertl and David, 2003). The optimization is done by duplicate each instruction body several times and uses different copies in different places of the code stream. In the extreme case, if every opcodes in the code stream can be duplicated, then each opcode will have its own jump location and can always be predicted correctly with BTB and threaded code.

Consider the example given in table 3.3 where prediction accuracy rate goes from 50% to 100% with the help of replication.

Code Stream	Prediction without replication	Code Stream with Replication	Prediction result with replication
start: A	A	*start:* A_1	A_1
B	goto	B	B
A	A	A_2	A_2
goto *start*	B	goto *start*	goto

TABLE 3.3. Replication with BTB and threaded code

However, if one chooses the inappropriate instruction to replicate, it can hurt the prediction as shown in 3.4; a sub-optimal instruction B is chosen to be replicated, where overall accuracy goes from 66.67% to 50%. To ensure for the best performance, one should replicate every opcode in the code stream by using dynamic replication, with careful analysis during code generation and use ‘memcpy’ to copy the body of the instruction similar as selective inlining.

Code Stream	Prediction without replication	Code Stream with Replication	Prediction result with replication
start: A	A	start: A	A
B	goto	B_1	goto
A	A	A	A
B	B	B_2	B_1
A	A	A	A
goto *start*	B	goto *start*	B_2

TABLE 3.4. Replication resulting in increasing mispredicitons

3.7 Other Related Work

3.7.1 Instruction Reordering

Brunthaler (Brunthaler, 2011) described an optimization approach by reordering the VM instruction in the instruction table, such that more frequent instructions can be successfully stored in cache level. This data-driven approach requires the writer to pre-analyze the instructions frequency in a particular VM and order the instructions accordingly. The author manage to show a factor of 1.142 speedups when experimented on Python using the *fasta* benchmark.

3.7.2 Code Generator for Interpreter

Another way to implement efficient VM is to use a code generator that automatically generates a stack-based interpreter for the programmer. Vmgen (Ertl et al., 2002) generates C code for the interpreter, just like how Yacc generates the parser. A significant advantage of using Vmgen is for the ease of implementation. More attracting, Vmgen applies complex optimization which usually hard for a programmer to write, such as super-instruction and replicate. In addition to that, Vmgen also supports scheduling and prefetching the VM instruction that targeting specific hardware architectures, such as PowerPC, which result in a 1.2 speed up. Experiment on GForth indicates that interpreter generated by Vmgen is faster and easier to implement than a hand-written interpreter, and only slower than the native code compiler by a factor of 2. However, machine-generated source code can be hard for a human to read and extend, and it adds the generator tool as an extra dependency for the interpreter implementation. Interpreter author may prefer to rely on the host language that has large community support from both academic and industry instead of a relatively small project with only a handful maintainers. That being said, the generated code from Vmgen can still be a good guideline for interpreter author to reference.

3.8 Modern Branch Predictor

The literature that is targeting on VM dispatch techniques all have one solid assumption: indirect jumps in VM interpreter are hard to be predicted. Although the assumption was valid given simple branch predictor back the time, the most recent research given by Rohou et al. (Rohou et al., 2015) offers a different opinion. They considered three interpreters (Python, Javascript, CLI) on a state-of-art branch predictors ITTAGE and three most recent Intel process generations in 2015. Their result indicates that the branch misprediction rate drop dramatically from 12 - 20 MPKI to an only 0.5 - 2 MPKI as the process evolved. In addition to that, the penalty of mispredictions has also decreased by a considerable amount — the number of instruction slots wasted due to branch misprediction drop from 14.5% on Sandy Bridge to 7.8% on Haswell.

Their finding may suggest that the dispatch optimisation techniques are no longer be effective on modern hardware. We were able to find few discussions about this paper in the language implementation community, but unfortunately, we were not able to find any formal research following up to this paper regarding the interpreter performance.

3.9 Chapter Summary

In this chapter, we introduced two interpreter architectures: tree-interpreter and the bytecode interpreter. Tree-interpreters like AST interpreter are usually considered to be slow but more straightforward to be implemented. Bytecode interpreter can be further broken down into two categories, stack-based or register-based. Many research suggests that register-based VM has better performance due to higher semantic density which leads to less dispatch in the execution. On the other hand, stack-based VM has a relatively smaller opcode size and is more convenient to implement. In addition, we survey the performance components of the interpreter, and we conclude the dispatch process, or branch prediction is the root cause of slowness in the interpreter. We explain in detail how branch prediction and pipeline scheduling affect the performance, and we survey techniques that are invented to overcome this issue. The techniques can be categorized into two kinds, one that focuses on writing dispatch process differently, such as threaded code and indirect threaded code; the other focus on redesign the instruction set, such as super-instruction and replica. Other techniques can help produce efficient interpreter including an interpreter generator or virtual instruction reordering. Finally, and most importantly, we found related research, suggesting the impact of a branch misprediction is no longer as effective as before as

hardware has improved recently. This brings us one of the critical questions in this work, how are those optimization techniques work on modern hardware and what is its performance on logic language like Soufflé. We will try to answer those questions in the later chapter with experiment and result.

CHAPTER 4

Switch-based Shadow Tree Interpreter

4.1 Overview

In this chapter, we introduce Switch-based Shadow Tree Interpreter (SSTI), a new implementation strategy that provides supreme efficiency and maintainability for tree-walk interpreters. The interpreter is tailored towards a C++ implementation - although it can be used for other languages as well. The SSTI strategy is similar to an AST interpreter. Hence, it targets on coarse-grained instructions instead of fine-grained instructions. This new design aims to overcome some of the engineering and performance limitations in AST interpreters while minimizing the software engineering overhead compared to more sophisticated implementation such as bytecode VM.

4.2 Challenges in AST Interpreter

An AST interpreter suffers from both software engineering concerns and performance degeneration with regarding runtime execution. The descriptive nature of AST fits awkwardly in an execution model, and the double-dispatches introduced by visitor pattern makes it suffers from dispatch overhead.

The visitor pattern is a common strategy to implement an AST interpreter in an object-oriented language. A visitor pattern in an interpreter is enabled by providing a virtual 'accept' function in the IR node and a virtual 'execute' function in the visitor (engine) class. The execution detail is shown in Figure 7, as mentioned in Chapter 3, two dispatches are involved. One comes from the IR node calling the visitor's 'execute' function, and the other comes from the visitor to dispatch the next operation node. The visitor pattern is easy to implement, and makes adding new operation in the language relatively easy. Furthermore, when language implementation has several execution strategies, such as a compiler and an interpreter, visitor pattern enables the language with different execution models without modifying the source IR; However, using visitor pattern to dispatch operations would double the dispatch costs

in the execution time, making a significant impact on the performance. In addition to that, AST fits awkwardly in the execution model when applied with runtime optimization. By principle, AST should not be modified during runtime as a purely descriptive format. Therefore, when applying pre-runtime optimization such as name encoding, runtime state cannot fit in the AST without violating the principle of separate responsibility. For example, when the AST is shared between multiple execution models, such as a compiler and an interpreter, no assumption should be made about how a particular model performs its own name encoding. A suboptimal solution is to provide a separate runtime data structure to handle runtime state, such as a hashmap that maps from tree node to runtime variable for name encoding. Such an approach introduces extra engineering overhead, and the performance of a hash map is inferior compared to direct index encoding with a list container (Muslija and Pjanić, 2018). A bold programmer may still be willing to violate a couple of engineering principles by putting pre-runtime optimization directly inside of an AST. However, when applying in-runtime optimization where nodes need to be modified on-the-fly (e.g. reconnect nodes, insert cache reference), directly modifying AST can be error-prone, breaking other parts of language implementation such as error reporting.

Listing 7 Double dispatch in visitor pattern

```
class RamNode{
    virtual accept(Visitor* visitor) {
        // visitor is a polymorphism type
        // calling the visitNode function requires a dispatch
        visitor->visitNode(this);
    }
}

class Interpreter : Visitor {
    virtual visitNode(RamNode* node) {
        /* Perform  regular execution  */

        // Dispatch the next operation.
        // node->child is a polymorphism type
        // calling the accept function requires a dispatch
        node->child->accept(this);
    }
}
```

4.3 Shadow Tree Overview

SSTI is specialized for light-weight switch-based interpretation for coarse-grained instructions. The implementation strategy expects the AST as input, and hence it relies on a recursive traversal pattern

through the tree. To make the execution efficient, a new tree data structure - shadow tree - is introduced, which shadows the shape of the AST. The shadow tree is an executable format generated from a tree-like IR. It overcomes many of the engineering limitations when implementing an efficient and maintainable tree-walk interpreter. First, compared to traditional AST interpreter, it enforces better engineering practices by separating descriptive format (e.g. AST) from the executable information. Therefore, higher-level IR can be well shared between multiple implementations. Second, it enables pre-runtime optimization by allowing information like name encoding to be inserted into the shadow tree node; it also allows in-runtime optimization by modifying the tree node on-the-fly during execution.

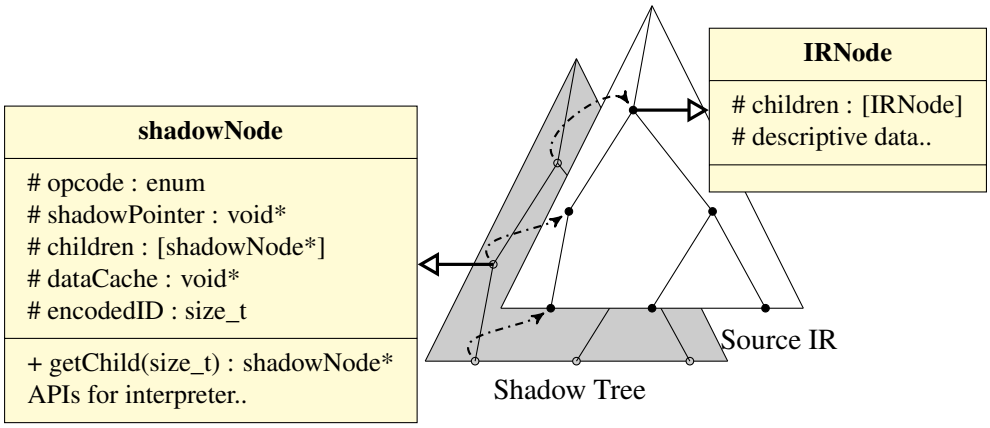


FIGURE 4.1. Shadow tree overview

Finally, shadow tree commits to a switch-dispatch pattern by holding an Enum variable in each node, saving half the dispatches in a traditional AST interpreter that relies on visitor pattern; using switch-dispatch also allows other dispatch techniques to be applied, such as indirect threaded code.

Figure 4.1 demonstrate the shadow tree structure. The shadow tree is generated from a tree-like source IR, the semantic of the shadow tree follows directly from the source IR. Therefore, the shadow tree has the exact same structure with its source IR [1], meanwhile, each shadow node holds a raw pointer (shadow pointer) points to the corresponding source IR node. Shadow node does not utilize class hierarchy. Instead, all nodes share the same base class structure and each node holds an Enum variable representing its operation type; child nodes are held within a list of shadow node pointers. In addition to that, the shadow node can also hold other attribute fields for storing runtime information on the need, such as data caching and name encoding.

[1] In practice, some nodes in source IR can be ignored based on need.

4.3.1 Why Tree Structure

Although bytecode representation can also overcome many of the issues of executing an AST directly, the shadow tree still has several advantages. First, compared to a bytecode representation, the shadow tree is much easier to implement. The instruction set comes for 'free' as it directly follows the source IR. Second, the effort of code generation is minimized by a single traverse of the source IR, while bytecode representation requires a more sophisticated code generation strategy, such as handling branch instruction, design appropriate instruction set. Third, the bytecode representation introduces fine-grained instruction. Although many imperative languages may favour it, it does not fit well when the language to emulate has a coarse-grained instruction set, such as Soufflé. The experiment about semantic density between bytecode representation and shadow tree is presented in Section 6, in which we found shadow tree spent 9% - 36% fewer dispatches compared to bytecode representation in Soufflé, results in fewer dispatches overhead. Finally, in a language implementation where the IR is already highly-optimized (such as Soufflé's RAM), it is doubtful whether it is necessary to design and implement a completely different format.

4.4 Execution Model

The interpretation is based on a recursive function call and switch-dispatch on the Enum variable within the tree node. The engine starts by dispatching the root shadow node. To enable execution, computation is dispatched to its corresponding virtual instruction by a switch dispatch. If any static information is required during runtime, it can be retrieved by looking thought the shadow pointer and cast it to the type of the corresponding source IR node. During the computation, the control is transferred to the child shadow node by following the semantic of the source IR using recursive call and switch statement.

4.4.1 Runtime Optimization

Separated from the static information, the shadow node is well compacted and can leave space for storing runtime related information. Runtime-related information can be inserted either during the generation of shadow tree or by directly modifying the shadow node during the runtime.

4.4.2 Example of Pre-runtime Optimization

Name encoding helps the interpreter to quickly identify the location of the target variable in the runtime environment. Although other representation can also utilize name encoding, the shadow tree has several advantages. First, putting name encoding in a higher IR violates the principle of separated responsibility, as we mentioned previously. Second, in a representation like bytecode, name encoding is easily achievable. Nevertheless, once the execution encounters any error, it is hard for a bytecode interpreter to report meaningful error message since the original name is lost during encoding and code generation. A separated error message reporter must be designed for decoding the variable name when a user report is needed in bytecode representation, which introduces even more engineering overhead. In shadow tree, name encoding can be done during the generation of a shadow tree, in case an error occurs and a report is needed, the original information can be easily retrieved by looking at the source IR through shadow pointer.

4.4.3 Example of In-runtime Optimization

In addition to that, the shadow tree also supports runtime optimization by modifying the shadow node on-the-fly during execution, such as data caching. Data caching may not be common in an imperative language, which focuses on light-weight and intensive arithmetic computation. While in a language like Soufflé, data caching of a runtime relation reference can save a significant amount of data overhead. In shadow tree, if optimization can be applied, the interpreter is free to modify the 'cache state' of a tree node and insert a caching reference. Then during the next execution, if the optimization state is valid, the interpreter can directly access the underlying data through the cache. Such optimization cannot be done with an AST representation because of the same engineering concern mentioned in the above section. On the other hand, bytecode representation requires extra spaces for arguments as the notion of 'cache state', and it has to modify the code stream on the stack during runtime. Such an approach is error-prone and leads to tedious code compared to the shadow tree, where the complexity of the algorithm can be well controlled with data attribute maintain in the tree node itself.

4.4.4 Summary

In summary, the shadow tree enables pre-runtime and in-runtime optimization, which is not possible or at least not considered as a good practice when using an AST interpreter. In particular, pre-runtime

optimization can be easily inserted into the tree node, and the error message can be recovered entirely by looking though the shadow pointer. Shadow tree allows optimization on-the-fly by modifying the tree node, which is simpler compared to modifying code stream on the stack.

4.5 SSTI or Bytecode Representation

Shadow tree tries to overcome many issues which can also be solved by bytecode representation. However, those two representations are fundamentally different, and both have its pros and cons depends on the situation. A careful decision should be made on whether bytecode or shadow tree representation should be used as the executable format.

First, the shadow tree can suffer from the recursive structure where a recursive function call transfers control flow, introducing overhead on the program stack. In comparison, bytecode representation is purely sequential and transfers control flow by changing the value of vPC. On the other hand, when emulating iterative statement, shadow tree can benefit from the native for-loop statement of the host language while a bytecode representation requires two or more low-level virtual instructions (condition check and goto). The save in the dispatch numbers can lead to a significant impact on the evaluation time when the target language is heavily iterative-based, such as Soufflé. In Soufflé, an iterative statement is based on high-level C++ iterator object; this gives shadow tree another advantage because sequential representation needs to read/store the iterator from/to the virtual environment repetitively, which is more expensive than just reading an integer value.

Second, by being a tree structure, shadow nodes are connected through pointers in the program, which is not cache-friendly when walking through the pointers. In contrast, bytecode representation is well compacted in a list container and preserve better data locality. However, bytecode representation is usually fine-grained and requires more dispatches while shadow tree is explicitly designed for emulating a coarse-grained instruction set, and its evaluation usually has fewer dispatches. The number of dispatches saved can amortize the impact of data locality depends on the language implementation. The trade-off should be considered when making an implementation decision.

4.6 Chapter Summary

SSTI as a new tree-walk interpreter overcomes the engineering concern in an AST interpreter by separating the executable format from the descriptive IR. A switch-based dispatch techniques reduces the dispatches cost by half and can be extended with other optimization techniques in the future. Its underlying implementation is simple compared to bytecode representation and the code generation is minimized by a single traverse in the source IR. In case of any runtime optimization is needed, author is free to insert extra information into the node or changing the internal structure of the trees without worrying about breaking engineering principles. The performance of SSTI can be compatible with bytecode representation based on the target language. In Soufflé, our SSTI implementation - Soufflé Tree Interpreter, is currently 5% - 10% faster then the bytecode representation - Soufflé Virtual Machine.

CHAPTER 5

Implementation of Soufflé

5.1 Overview

This chapter gives a detail description about the implementations of the Soufflé interpreter. Two interpreters are designed and implemented, a tree-walk interpreter utilizing SSTI strategy called Soufflé Tree Interpreter (STI); and a stack-based VM with switch dispatch called Soufflé Virtual Machine (SVM). In addition to that, a data structure adopter is implemented in order to support the runtime interpreter with statically typed, efficient and parallel data structures.

5.1.1 Challenges and Legacy Implementation

Before this work, Soufflé has a legacy AST interpreter which directly executes on the RAM IR. In order to share RAM between the C++ synthesizer and interpreter, the execution is enabled by a visitor pattern. As a result, the dispatches are doubled during execution and causes a performance degeneration; and interpreter specific runtime optimization is tedious to implement, requires much engineering effort.

The legacy interpreter also suffers from the inefficient runtime data structure. Soufflé's outstanding performance relies on the parallel, efficient data structure. The data structure is statically typed, hence the initialization of the data structure requires knowledge of source Datalog program, such as the arity size of the relation. A synthesizer can produce C++ code to initialize those data structure based on the input program and compile them into binary code. However, there is no opportunity for an interpreter to initialize static data since the program evaluation starts as soon as RAM is generated. As a result, the legacy interpreter uses an alternative runtime container, erase the static type of tuple into raw pointers and emulate the same functionality. Nevertheless, the approach introduces memory and performance overhead and does not support parallel execution, making the interpreter scale poorly on the real-word sophisticated program.

Part of this work is to provide an efficient interpreter implementation for Soufflé with support of parallel execution and static data structure. Meanwhile, good software development principles need to be established in order to provide maintainable and extendable code.

5.1.2 Soufflé Tree Interpreter

STI is implemented using the Switch-based Shadow Tree strategy mention in Chapter 4. Soufflé translate its source program into RAM - a tree-like, imperative-style intermediate representation. The shadow tree takes the structure of RAM representation, within each shadow node there is an Enum variable indicating the operation type; a shadow pointer referencing to the source RAM node; a list of shadow nodes maintaining the tree structure; a list of 4 bytes integer for operation arguments; and finally a raw pointer for runtime relation caching. The generation of the shadow tree is done by utilizing the visitor pattern; the shadow tree generator is passed into the 'accept' function into the root of the RAM tree to fire up the generation. An example of generating a shadow node that points to a `Scan` operation is shown in Listing 8. Using visitor pattern for code generation double the dispatch costs, however, the code generation only needs one pass through the tree, and the extra overhead can be amortised once the interpreter starts the execution using switch-dispatch.

The execution is enabled by switch statement and the control flow in transferred into child node by recursive tree traverse. Listing 9 demonstrate the virtual instruction of a `Scan` operation. Note that, although shadow node holds a reference pointer to the source RAM node, it is not always required to reference the source for execution. For example, executing a `Scan` operation does not requires referencing the source `RAMScan` node. However, when an error occurs, we can report a detailed log message by casting the shadow pointer into a pointer to `RAMScan` and retrieve any necessary information, such as target relation name, its surrounding operations, etc. Depends on the target operation, we may or may not want to cast the shadow pointer for execution purpose. To avoid redundant and error-prone coding, we create a macro to help declare a case statement. *CASE(Kind)* is used for operations where no casting is required for execution while *CASE_CAST(Kind)* will cast the shadow pointer for execution purpose as soon as the program enters the case statement. Finally, *ESAC(Kind)* helps us make sure every case statement is ended properly.

In addition to the basic structure and execution model, we have implemented pre-runtime and in-runtime optimization directly in the tree node by taking advantage of the shadow tree. In the following sections,

Listing 8 Example of generating RamScan operation

```
using NodePtr = std::unique_ptr<ShadowNode>;
using NodePtrVec = std::vector<NodePtr>;
class ShadowTreeGenerator{
    NodePtr visitRamScan(RamScan* scan) {
        // Encode operation hint as index id
        std::vector<size_t> data;
        data.push_back((encodeHint(&scan)));
        // Location to store target tuple
        data.push_back(scan.getTupleId());

        NodePtrVec children;
        // Generate child nodes.
        for (auto child : scan.getChildren()) {
            children.push_back(child->accept(this));
        }
        // Operation_Scan is the Enum variable.
        // &scan specify the referenced source IR.
        // data is the operation argument.
        return std::make_unique<ShadowNode>(Operation_Scan,
            &scan, std::move(children), data);
    }
}
```

we describe how runtime optimization works in STI implementation and how to reduce the cost of register allocation in a recursive switch-dispatch execution model.

5.1.2.1 Name Encoding

Name encoding helps interpreter to locate the target variable in the runtime environment quickly. In Soufflé we perform name encoding on Relation Operation Hints (ROH) (Jordan et al., 2019b), which is a cache object containing the latest query information in the target relation. Therefore any following operations can make use of the data ordering properties and boost up the performance. ROH is bound with a specific relation operation, which is a node in the shadow tree. By definition, ROH can only be created during the runtime and needed to be updated after each subsequent execution. In legacy implementation, due to the engineering principles of not inserting runtime information into RAM directly, the encoding is done by a hashmap that maps from the node address to the location of ROH. In STI implementation, tree node is encoded with a unique id during shadow tree generation, and the id is provided as operation arguments during the execution, the example in Listing 8 includes name encoding procedure of ROH. Finally, ROH is stored in a list container where its position is the operation id produced during code generation. The number of hints needed to be created is known before runtime, hence the memory

Listing 9 Virtual instruction of Scan operation

```
#define CASE(Kind) case (Operation_##Kind): {
#define CASE_CAST(Kind) \
    case (Operation_##Kind): { \
    auto source = static_cast<Ram##Kind>(node->source);
#define ESAC(Kind) \
    fatal("Operation " #Kind " finished without returning result");\
    }

// Note ctxt is the virtual environment for storing temporary variable
Result execute(ShadowNode* node) {
    switch(node->opcode) {
        CASE(Scan)
            size_t relId = node->getData(0);
            auto& rel = ctxt.getRelation(relId);

            // use simple iterator
            for (const RamDomain* tuple : rel) {
                ctxt[node->getData(1)] = tuple;
                // Transfer control through recursive function call
                if (!execute(node->getChild(0), ctxt)) {
                    break;
                }
            }
            return true;
        ESAC(Scan)
    }
}
```

layout of the container is fixed once the program starts executing; the only operations required on the container are the *read* and *write*; compared to legacy implementation, access hints with direct index has better performance than using hashmap (Muslija and Pjanić, 2018).

5.1.2.2 Relation Caching

Based on the static nature of RAM, the relations need to be created during runtime are known before program execution. This means relations can be statically encoded into index id just as operation hints. However, relation referencing are very frequent in RAM operation. Therefore, we want to save an extra indirection from list indexing by holding a reference to the target relation directly. To achieve this, we perform relation caching on-the-fly during runtime. For each operation node, the first time it accesses the relation would be by list indexing; once the address of the target relation is known, it will be inserted into the tree node. The static nature of RAM ensures that the caching will be valid throughout the program

lifetime. As a result, any subsequent access can be done by directly go through the reference, saving us one level of indirection. The resulted code after apply relation caching in the `Scan` operation is shown in Listing 10.

Listing 10 Example of applying relation caching

```
CASE(Scan)
    if (node->cache == nullptr) {
        size_t relId = node->getData(0);
        auto& rel = ctxt.getRelation(relId);
        node->setCache(&rel);
    } else {
        auto& rel = *node->cache();
    }
    /* Other computation .. */
ESAC(Scan)
```

5.1.2.3 Reduce Register Pressure In Recursive Function Call

To transfer program control, the interpreter calls an 'execute' function with the child node as an argument. In the assembly level, once enter the new function body, the program needs to allocate a new stack frame and push all the callee-saved registers onto the stack to prevent the instruction in the function from overwriting those registers' value. The compiler determines what register to save by looking at the instructions of the function body. However, the 'execute' function in STI contains only a long switch statement, and any of the cases is a possible target to execute during runtime. As a result, the compiler would decide to save the maximum possible number of registers needed for the heaviest case statement. Using GCC 9.0 with -o2 or -o3 option targeting on a x86_64 architectures produces the following code (Listing 11) when calling 'execute'.

Assembly code spends six instructions to save the callee-saved registers, which is not needed by many virtual instructions. In addition to that, it spends three instructions to create a canary value to prevent stack buffer overflow attack (Bryant and O'Hallaron, 2015). However, this is not always needed since many of the virtual instructions does not allocate a buffer on the stack.

To work around this, we put each of the virtual instructions inside of a local C++ lambda - warping it as a function call. This can be achieved easily by modifying the macro mentioned in Listing 9. Therefore, the compiler will be noted that no callee-saved register is needed and choose not to overreact (Listing 12).

Listing 11 Assembly code to prepare a function call

```
## Allocate program stack
push    %rbp
mov     %rsp,%rbp

## Save callee-saved registers
push    %r15
push    %r14
push    %r13
push    %r12
push    %rbx
sub     $0x868,%rsp

## Produce Canary value to prevent stack overwrite
mov     %fs:0x28,%rax
mov     %rax,-0x38(%rbp)
xor     %eax,%eax

## Then start the switch statement
```

Instead, the compiler would save registers accordingly once enter the lambda. This optimization produces the assembly code as shown in Listing 12, it merely allocates the stack and starts the switch right away, saving nine instructions.

Listing 12 Assembly code after optimization

```
## Allocate program stack
push    %rbp
mov     %rsp,%rbp

## Start the switch statement
```

The trade-off here is the extra function call; however, based on our experiment, the save in instructions justify the extra function call, saving us 5% - 12.5% instructions based on the benchmark. This is because light-weight virtual instructions which do not need any callee-saved registers are executed much more frequently than heavy instructions in RAM.

5.1.3 Soufflé Virtual Machine

Soufflé Virtual Machine (SVM) is a stack-based Virtual Machine Interpreter and has its own intermediate representation – a bytecode representation. The data stack is implemented with a `std::vector<32_int>` container, because all the operations in Soufflé are expected to return a single 32-bit integer. SVM has

98 instructions, each encoded as a standard `Enum` in C++, which is equivalent to an `Int` and results in a 4 bytes opcode size on a 64-bit machine.

Unlike STI, SVM instructions are fined-grained, consists of a few numbers of simple virtual instructions, such as `SVM_LOAD`, which loads a tuple onto the stack; and `SVM_Goto`, which modify the value of vPC. Iterative-based instructions are cut down into several fine-grained operations as well. For example, `Scan`, which iterates through all the tuple within a relation, is divided into three operations: `SVM_Init_Scan`, `SVM_Read_Iterator` and `SVM_Goto`.

- `SVM_Init_Scan` takes two arguments, one for target relation and one for the position to store the iterator. It initializes a full range iterator from the target relation and stores it into the target position in the virtual environment.
- `SVM_Read_Iterator` takes two arguments, one for the target position of the iterator and one for the alternative branch location to jump in case the iterator is exhausted. It reads the iterator from the virtual environment, reads its value, increments the iterator and pushes the value onto the stack. In case the iterator is invalided, it jumps to the alternative location by modifying the vPC value.
- `SVM_Goto` takes one argument; it merely jumps back to the start of the loop.

Such fine-grained instruction set helps the SVM to maintain a sequential execution model and does not require recursive function call as in STI, therefore, it avoids the extra program stack overhead. However, it is worth to mention that we do not tend to emulate every single low-level instruction in SVM, as it can cause dispatch overhead and performance slow down. Instead, we seek every opportunity to rely on native support from the host language to reduce the overhead of interpretation; for example, not all the conditional branch is implemented as an SVM instruction, if suited, we will save the extra dispatch by implementing them as a native if-statement in C++. As an example, Listing 13 shows the implementation of `SVM_Read_Iterator`, notice how the branch is implemented in native if-statement in C++ instead of breaking into virtual branching such as `SVM_Jumpez`.

SVM utilize the same runtime optimization as in STI. Hints are encoded as id and provided as operation arguments. The relation is statically encoded as an index as well because memory is managed by smart pointers, which cannot be freely cast into integer type and provided as operation arguments directly. Therefore, SVM spends one extra indirection when accessing relation through the container.

Listing 13 Implementation of Scan in SVM

```
// vPC is the virtual instruction pointer
// Ctxt is the virtual runtime environment
Result execute(){
  while(vPC){
    switch(codeStream[vPC]){

      case(SVM_Read_Iterator){
        // Retrieve iterator from environment
        size_t iteratorId = codeStream[vPC+1];
        auto& iter = ctxt.getIterator(iteratorId);
        if (iter.notEnd()) {
            stack.push_back(*iter++);
        } else {
            // If iterator is exhausted, jump to alternative branch.
            vPC = codeStream[vPC+2];
            break;
        }
        vPC += 3;
        break;
    }

  }
}
```

5.1.4 Data Structure Adapter

The data structure in Soufflé provides efficient performance for relational algebra operations by exploiting the static information from the source Datalog program such as relation (arity) and specialized comparator (indexing) (Subotić et al., 2018). However, the interpreter is not benefited from it as it cannot initialize objects with static information during runtime. An adapter is implemented to provide uniformed APIs, warping around the static classes for the interpreter. In detail, the static classes that need to be uniformed by the adapter are:

- **Tuple** Tuple is the basic unit stored in a relation. The size of the tuple is statically typed (arity).
- **Comparator** A comparator defines a total ordering in a relation. A comparator uses the variadic temple in C++ (Lippman et al., 2012) and can take a variable number of arguments indicating the order of elements to compare. For example, an order $< 0, 2, 1 >$ indicating that the first element should be compared first, if equivalent, then compare the third element and finally the second element. Comparator enables relation range query and existence check with excellent performance in Soufflé.

- **Data Structure** To initialize a data structure, the user has to statically provide the tuple type (arity), comparator (order) and the underlying data structure implementation. Soufflé currently has five different underlying implementations for data structures, a Nullary for tuples with zero arity, a B-tree (Jordan et al., 2019b) and Brie (Jordan et al., 2019a) for generic usage and an Eqrel structure (Nappa et al., 2019) for equivalence relation with binary elements. In addition to that, B-tree has a provenance version (Zhao et al., 2020) that is used for provenance evaluation.

5.1.4.1 Adapter Overview

The adapter aimed to provide uniformed interfaces for interpreter regardless of the underlying static information. The idea is to force the compilation of the template class during the building process of the interpreter, and warping them with uniformed interfaces, hence objects can be created and used during runtime. For a quick example, imagine a case where the user would like to create a `std::array<T, size_t>` with variable length during runtime and want to be able to modify the value of the array regardless of its static type. Listing 14 demonstrate how to achieve it, by forcing the compilation of `std::array<int, size_t>` and build up a factory such that object with static information can be created during runtime. In addition to that, the `base()` function in the base class provide a type-erased way for user to modify the underlying values. We used a similar approach to virtualize the relation arity and underlying data structure implementation. Currently, Soufflé interpreter support arity up to 30, this results in 89 classes (29 each for b-tree, brie, provenance (b-tree), one for Nullary and one for Eqrel). Although such an approach does not support arity with an arbitrary size, this is not a major concern in Soufflé for most of the user cases. Based on our experience, we believe the current limit is sufficient for most cases, and we have not had any application requires arity larger than the current limit. In case the user really needs a larger arity, he or she is free to modify the source code and re-build the interpreter for their needs. The trade-off of this approach is the extra virtualized interfaces in the base adapter - any usage from the adapter would result in a virtual dispatch.

5.1.4.2 Virtualize Comparator

Above approach does not completely solve our problem - we still have comparator to virtualise. However, the comparator can define any possible order of underlying tuple, which results in $O(n!)$ number of possible orders for relation with an arity of n. Therefore, we can not simply list all permutations.

Listing 14 Example of producing statically typed object with factory and adapter

```
// Base class for providing virtual interface for a list of Int
class ListAdapter {
public:
    virtual int get(size_t) = 0;
    virtual int* base() = 0;
};

// A ListAdapter with underlying implementation std::array
template <size_t Length>
class IntArray : public ListAdapter{
public:
    virtual int get(size_t i) override {
        // Bound check omitted for simplicity.
        return array[i];
    }
    // Return a raw pointer, such that user can use
    // the underlying data regardless the static type.
    virtual int* base() override {
        return array.data();
    }
private:
    std::array<int, Length> array;
};

// A factory that can return IntArray based on runtime information
std::unique_ptr<ListAdapter> intArrayFactory(size_t length) {
    switch(length) {
        case(0): return std::make_unique<IntArray<0>>();
        case(1): return std::make_unique<IntArray<1>>();
        case(2): return std::make_unique<IntArray<2>>();
        ...
        case(30): return std::make_unique<IntArray<30>>();
        default: fatal("Size not support yet.");
    }
}
```

Instead, we use the default comparator order for all data structure - an ascending sequence ordered from 0 to $n-1$. Then, before a tuple is inserted into the relation, we rearrange its elements by the actual desired comparator order, such that the resulting final order in the comparator is equivalent (Figure 5.1 and Figure 5.1). Formally speaking, for a data structure with arity n, denote the natural order, or the identity permutation id be $(0, 1, 2, ..., n-1)$, and id will be used as the comparator for the relation instead. Let the actual desired comparator order for the data structure be ϕ. For any tuple $t = \{x_0, x_1, ..., x_{n-1}\}$ need to be inserted into the data structure, we insert $\phi(t)$ instead. It is trivial to see the resulting order

for comparison is the same, $\phi(id(t)) = id(\phi(t))$. Similarly, when reading tuple t' from the relation, we apply the inverse permutation ϕ' to get back the original tuple, $t = \phi'(t') = \phi'(\phi(t))$. The inverse permutation is obtained by simply reverse the order of elements in ϕ. As a result, we can provide comparator during runtime; the trade-off there is the extra encoding and decoding when writing and reading the tuple.

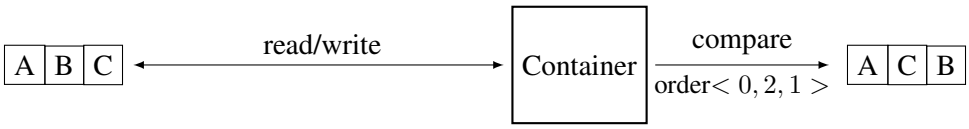


FIGURE 5.1. Tuple IO in static version (Synthesiser)

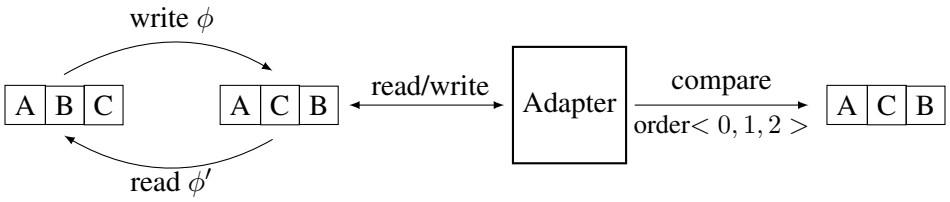


FIGURE 5.2. Tuple IO in dynamic version (Interpreter)

5.1.4.3 Unify Iterator

The last step is to support a uniformed iterator, such that data structure with different underlying implementations (such as Nullary v.s. Btree) can be iterated regardless of their static information. The challenges here is that, iterators can have very different behaviour based on the underlying data structure implementation. For example, Nullary relation is either empty or contains an empty tuple '*()*'; internally, it is a boolean value rather than a container. Such differences in the data structure indicate that we have to virtualise the behaviour of the iterator. Nevertheless, a Soufflé program can easily have millions (Jordan et al., 2020) of iterator-related operations; virtualising iterator would results in serious virtual dispatch overhead as every read/increment operation in the iterator will become a virtual function call. To overcome this, we design `Stream` - an abstract perspective on a data and `Source` - a virtualized interface for data source.

Each `Stream` is associated with a `Source`, and data is read through virtualised interface from `Source` instead of directly from the data structure. `Stream` can then be used as an interface and provides a uniformed, non-virtualised implementation for iterators. However, when `Stream` reads from `Source`, it still has to rely on the virtual interfaces. To minimise the virtual overhead, instead of reading from

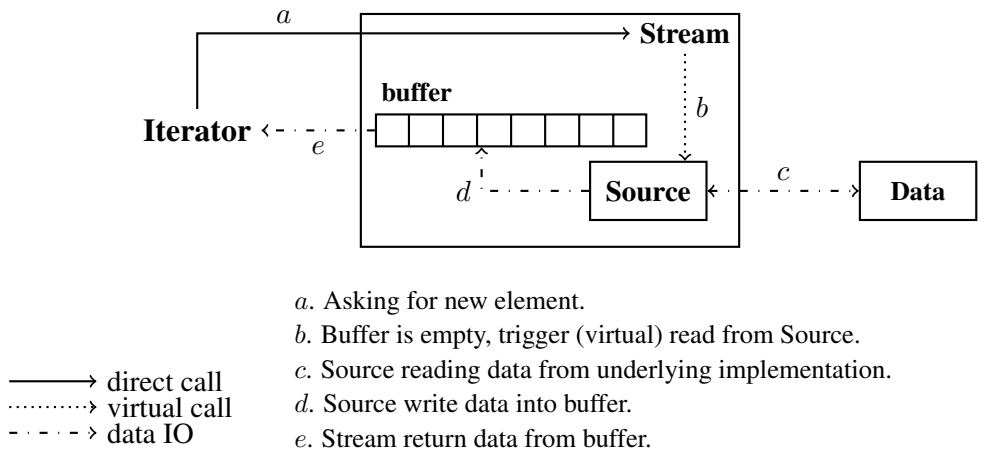


FIGURE 5.3. Diagram of Stream and Source mechanism

`Source` every time user increment and dereference the iterator, `Stream` reads more than one tuples from the `Source` at once and store them in an internal buffer. The current buffer size is defined to be 128, which means when the buffer is empty, the first read request would trigger `Stream` to read 128 tuples from `Source` at once, through a *single* virtual call. As a result, for the following 127 read requests, `Stream` directly returns values from the buffer instead of asking from `Source` through virtual interface, as shown in Figure 5.3. In brief, by utilising an internal buffer, we can amortise the virtual dispatches costs when iterating though the underlying data structure.

5.2 Chapter Summary

In this chapter, a data structure adapter is presented as a way of deploying an efficient static data structure during runtime for the interpreter. First, we show the technique of how to create a statically typed data container with runtime information by limiting the relation size. Second, we virtualise the comparator by tuple reordering for reading/writing during runtime. Finally, we present an efficient adapter with amortised virtual dispatches costs to support virtual iterator behaviour.

Two interpreter implementation are given, STI - a Switch-based Shadow Tree Interpreter shadowing on the RAM representation, and SVM - a stack-based switch dispatch VM based on the SVM bytecode representation. STI focuses on coarse-grained instruction set with recursive execution model while SVM commits to a sequential execution model by using a fine-grained instruction set. We also demonstrate some examples about how to implement runtime optimisation on those two implementations. In the next

chapter, we present performance analysis on the two interpreters and give a detailed explanation about their pros and cons.

CHAPTER 6

Experiment and Evaluation

Throughout this work, we present the implementation of two interpreters, STI and SVM. The two interpreters pursue different intermediate representations and different execution models. We must understand the performance difference between those two; more importantly, answer the root causes of the difference. In addition to that, we would like to experiment with different optimisation techniques on dispatch and instruction set, carefully examine their performance on modern hardware.

In this section, we aim to answer the following research questions:

(1) Which Soufflé implementation is faster and what is the performance ratio between the interpreter mode and the synthesiser? Did we build an efficient interpreter for Soufflé?
(2) For whatever interpreter implementation that is faster, what are the reasons behind it? Can we learn from the superior one and improve the inferior one, potentially make it better?
(3) As modern hardware improved, is dispatch/instruction optimisation still as significant as it was? In particular, what is the effect of indirect threaded code dispatch on modern hardware and how much performance boost can it bring to Soufflé's interpreter.
(4) What are the bottleneck components in Soufflé interpreter? Can we build a performance model for the interpreter to explain the gap between synthesiser? Can this performance model leads to guided optimisation decision and benefit the interpreter?

Throughout the experiment section, our evaluation will be mainly based on the result of following benchmarks:

- *Virtual Private Cloud (VPC)* benchmark (Backes et al., 2019). A real-word, network reachability reasoning tool, which provides a security-oriented solution in Amazon Cloud Service. Soufflé is part of the logic reasoning engine in VPC.

- *Ddisasm* (Flores-Montoya and Schulte, 2020) is a disassembler for reassembling the assembly code. The disassembler engine is implemented in Datalog and can be executed by Soufflé; it takes input a binary and produces the reassemble assembly code with proper symbolic information. For the experiment, a set of facts is extracted from the *SpecCPU2006* benchmarks and is feed into *Ddisasm*.
- *Doop* (Bravenboer and Smaragdakis, 2009) is a general and fast framework for the static points-to analysis of java program that produces precise analysis including context-sensitive, context-insensitive, call-site sensitive and objects sensitive analysis. For this experiment, we use the *1-object-sensitive+heap* analyses along with *Doop* benchmark sets.
- *tc* is a standard Datalog benchmark built on first-order logic for calculating the transitive closure of the input data. The benchmark is chosen for its simplicity and intensive computation which provides us with a controllable benchmark that scales well with input size.

All benchmarks are run multiple times on an Intel Xeon Gold 6130 CPU @ 2.10Ghz, with the governor of the CPU frequency set to *performance* to ensure the consistency. *PAPI* (Terpstra et al., 2010) is chosen as the profiling tool. It is a high-performance profiling application that provides the user with a set of standard and easy-to-use interfaces for general-purpose profiling and hardware event performance measurement.

6.1 Performance Measurement on SVM and STI

In this section, we evaluate the performance difference among different implementations. For the synthesising mode, we separate the runtime and compilation time since most of the use cases of the synthesiser is to compile the program for once and execute analysis on different input with the same binary.

Based on the result shown in Figure 6.1, the synthesiser has a clear advantage over both interpreter implementations, approximately 2.12 - 5.88 times faster than the STI.

In addition to that, STI has a slightly better performance compared to the SVM. This would not be expected in a typical language implementation as most of the language engineers believe a VM interpreter would be faster than a tree-like interpreter due to a more compressed representation.

The performance difference can be explained from the perspective of instruction set design. The instruction set we defined for SVM does not fit well with respect to the semantic of the RAM instruction

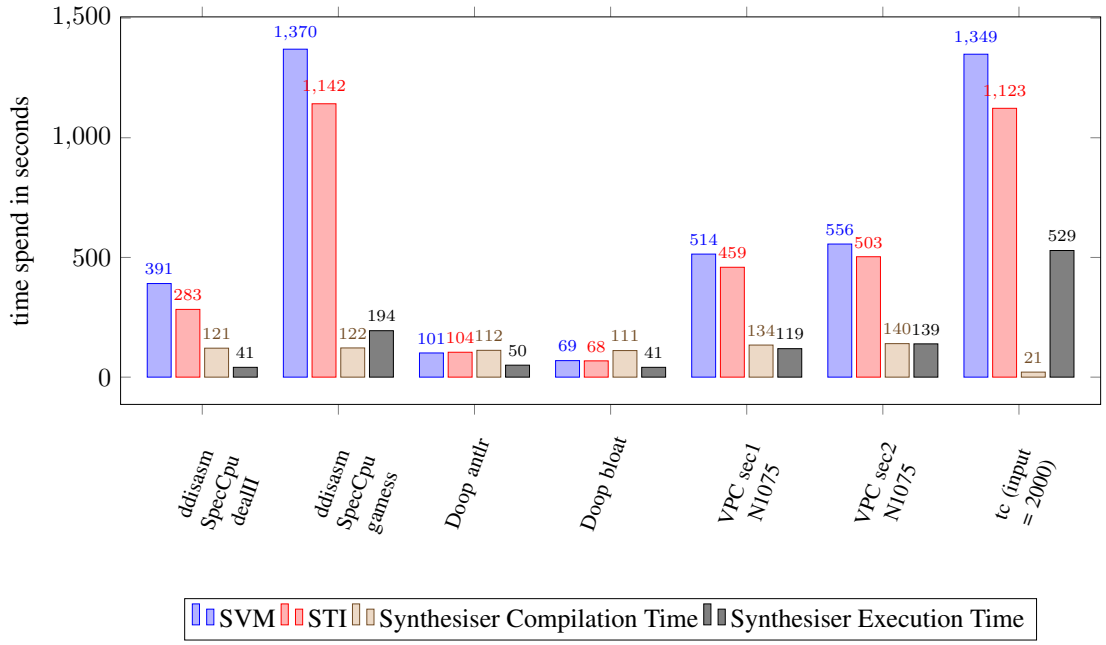


FIGURE 6.1. Performance evaluation on different implementation

set. In particular, SVM uses a fine-grained instruction set where the program is broken down into many small and light-way virtual instructions while STI uses a coarse-grained instruction set where operations are heavy-weight. The difference in semantic results in a difference in dispatch numbers, intuitively, a fine-grained machine would require more dispatches to finish the same computation compared to a coarse-grained machine. In the following experiment, we review the semantic nature of RAM and analysis the impact of instruction set on the resulting dispatches number.

6.1.1 Semantic Density

In Soufflé, the source datalog program is translated into RAM IR, which is a language describing the evaluation of the source datalog imperatively. RAM is heavily iteration-based; range query and tuple filtering are translated into an iterative statement on the target relation.

For example, Figure 6.2, 6.3 demonstrate how RAM describing the evaluation of a left-transitive closure. In RAM representation, the new knowledge is generated through two nested for loop, a `Scan` at line 3 walks through all the tuples in the `delta_tcl`, and `IndexScan` at line 4 performs a range query and goes through all the tuples in `base` that follows the condition. Take `Scan` as an example, assuming a `Scan` operation is performed on a relation with n elements with a single nested operation; in STI, the iterative statement will be translated into a native for-loop in C++, resulting in only one dispatch

```
.decl tcl  (x:number, y:number) output
.decl base (x:number, y:number) input

// Left-Recursive Transitive Closure
tcl(X, Y) :- base(X,Y).
tcl(X,Y) :- tcl(X,Z), base(Z,Y).
```

FIGURE 6.2. Source program

```
1  Loop
2    IF (((delta_tcl != ∅)) AND ((base != ∅)))
3      FOR t0 IN delta_tcl
4        FOR t1 IN base ON INDEX t1.0 = t0.1
5          IF ((t0.0,t1.1) ∉ tcl)
6            PROJECT (t0.0, t1.1) INTO new_tcl
7    EXIT (new_tcl = ∅)
8    FOR t0 IN new_tcl
9      PROJECT (t0.0, t0.1) INTO tcl
10   SWAP (delta_tcl, new_tcl)
11   CLEAR new_tcl
12 END LOOP
```

FIGURE 6.3. RAM representation of the main evaluation body

for the entire loop and n dispatches for the nested operation in the loop (Figure 6.4). While in SVM, each iteration is broken down into two virtual instructions. A `SVM_Iterator_Read` at the start of the loop, reading the value from the iterator, and branching to an alternative location if the iterator is invalid. A `SVM_Goto` at the end of the loop to jump back to the start of the statement. Therefore, each iteration requires two dispatches, results in $2n$ dispatches for the entire loop and n dispatches for the nested operation (Figure 6.5). For this example, the fine-grained instruction design in SVM produces 2 times more dispatches then STI with a coarse-grained instruction set.

To further confirm the hypothesis, we measured the operation density of the SVM and STI instruction set. The density is defined by the number of machine instruction per dispatch. Hence a higher density design would lead to fewer dispatches in the interpretation. The profiling is done by adding *PAPI* function (`PAPI_TOT_INS`) to count the number of machine instruction executed during the interpretation; in a separate run, we also track the number of high-level dispatches in the interpreter level; finally, density is obtained by calculating the average number of machine instructions for each interpreter dispatch. We expect that SVM would have a lower density which leads to a higher total number of interpreter dispatches compared to the STI.

```
// loop only require one dispatch
case(Operation_Scan) {
  // Total dispatches: 1 (loop) + n (child) = n
  for (auto& tuple : delta_tcl) {
    /* nested operation */
    execute(node->child);
  }
}
```

FIGURE 6.4. Evaluation of iterative-statement in STI

```
// loop require 2 dispatch per iteration
// Total dispatches: 2n(loop) + n(child) = 3n
1 SVM_Iterator_Read delta_tcl 4
2 /* nested operation */
3 SVM_GOTO 1
4 ...
```

FIGURE 6.5. Evaluation of iterative-statement in SVM

The result of the experiment is shown in table 6.1. In three out of four benchmark sets, SVM tends to have a more substantial number of dispatches but lower number of machine instructions per dispatch, which is what we expect based on the design of the SVM instruction set. However, in *tc* benchmark, SVM seems to also have a higher number of machine instructions per dispatch as well. Based on the RAM representation of *tc* (part of it is in Figure 6.3), which mostly consists of for-loop operation with only a few other basic RAM operations. The measurement result of *tc* should be more relevant to the real efficiency of the emulation of iterative statements then other complex benchmarks.

Given that SVM spends more instructions per operation code, we conclude not only emulating the for-loop causes more operation dispatches in the SVM, it can also lead to a performance inefficiency when compared to using a native C++ for-loop statement to traverse the data structure. We believe the inefficiency comes from the extra cost for the SVM to fetch the iterator in a separated virtual container every time it reads an element. In contrast, STI uses native C++ for-loop statement, where the iterator is likely to be stored in a machine register or at least on the program stack directly.

6.1.2 Summary

In summary, the performance ratio between synthesiser and the interpreter indicating that we have managed to implement an efficient interpreter. In addition to that, we observe our STI outperform SVM

on all benchmarks in the current implementation, which indicate a tree-walk interpreter can still have better performance than a bytecode interpreter based on the situation. In particular, the performance advantage comes from a better instruction set design that fits well in the RAM semantic. The heavily iterative-based nature of RAM decides that a fine-grained machine (SVM) would spend too many virtual dispatches, whereas a coarse-grained machine (STI) take advantage of native for-loop support from the host languages ends up saving dispatches, becoming more performance efficient. Finally, in order to better fit the SVM with RAM semantic, either the instruction set has to be redesigned (by giving up the flat execution model), or the runtime environment should be improved to support SVM with faster iterator reading.

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%

TABLE 6.1. Operation density

6.2 Investigation on Indirect Threaded Code Optimization

In Chapter 3, we mentioned that the ITC was a very popular optimisation technique targeting on reducing branch misprediction. However, in 2015, an experiment from Rohou et al. (Rohou et al., 2015) suggested that, with the modern branch prediction became more powerful, and prediction penalty becomes less critical, the impact of indirect threaded code dispatch may have become less critical. Based on their result, the branch misprediction rate drop from 12 - 20 MPKI to 0.5 - 2 MPKI as the processor evolve, and the number of instruction slots wasted due to misprediction drop to 7.8% on Haswell. Despite

the new finding, nowadays, there is still plenty of interpreter implementations that utilise the ITC. For example, by the time of writing this thesis, we can still find a comment in CPython's current source code (Version 3.9.05) which can be traced back to 2002, stating that:

> At the time of writting, the "Threaded Code" [1] version is up to 15% - 20% faster then the normal "switch" version, depending on the compiler and the CPU architecture.

To have a better understanding of indirect threaded code on modern hardware, we set up experiments on CPython comparing the performance difference. We also implement a version of SVM using indirect threaded code; we will measure its performance difference as well. We would like to determine whether it is necessary to apply the optimisation on Soufflé based on the performance impact.

6.2.1 CPython Experiment Setup

The version of CPython used for this experiment is 3.9.05, and the benchmark framework used is *pyperformance* (Stinner, 2020), which is an open-source project built as an authoritative source of benchmarking Python implementations, focusing on the real-world benchmark.

In CPython source code, it uses a macro to switch between indirect threaded code and switch dispatch as demonstrated in Figure 6. We alter the macro and produce two different binaries for our experiment.

6.2.2 CPython Experiment Result and Evaluation

As shown in Figure 6.6, the runtime of the switch dispatch version is used as a baseline. We can see that the threaded code dispatch still has a performance advantage over the switch dispatch method. However, the improvement in performance is no longer as pleasant as stated in the comment. 2 out of 17 benchmarks have a performance equal to the switch dispatch, and switch dispatch is actually being faster in 3 of the benchmarks, probably due to ITC having an extra layer of indirection during dispatching. For the remaining benchmarks, although ITC is still faster, most of them only have a performance difference about 5% -14% comparing to 15% - 20% which is stated in the source code comment.

[1] By looking at the source code, we believe the author of the comment was actually referring to "indirect threaded code"

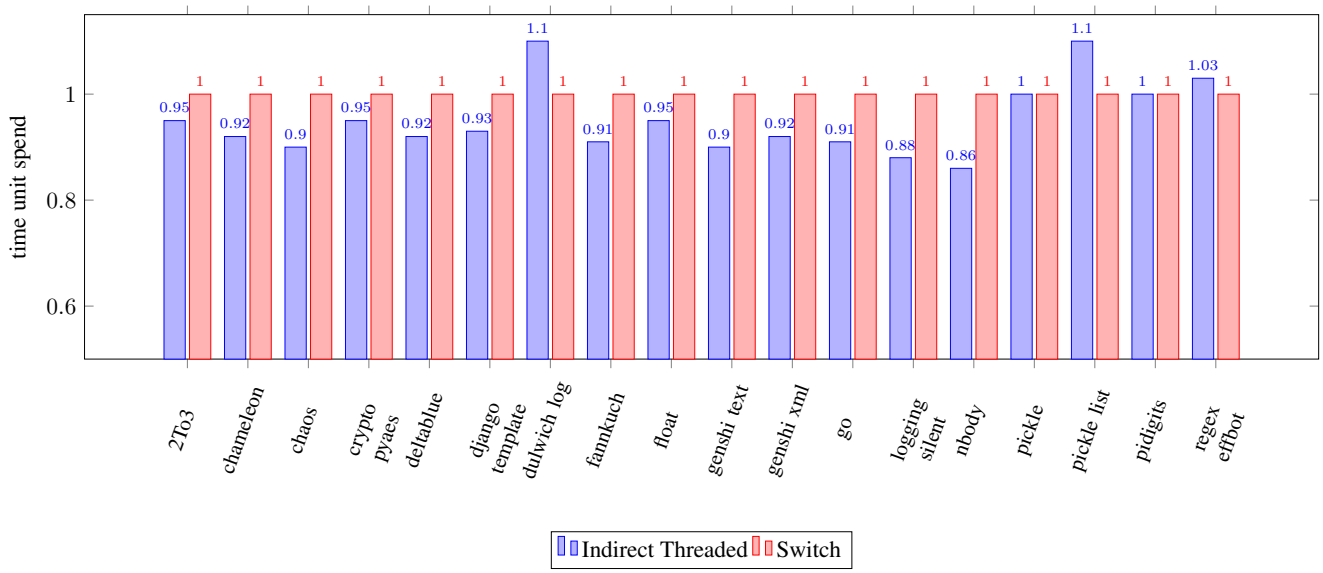


FIGURE 6.6. Cpython performance using different dispatch methods

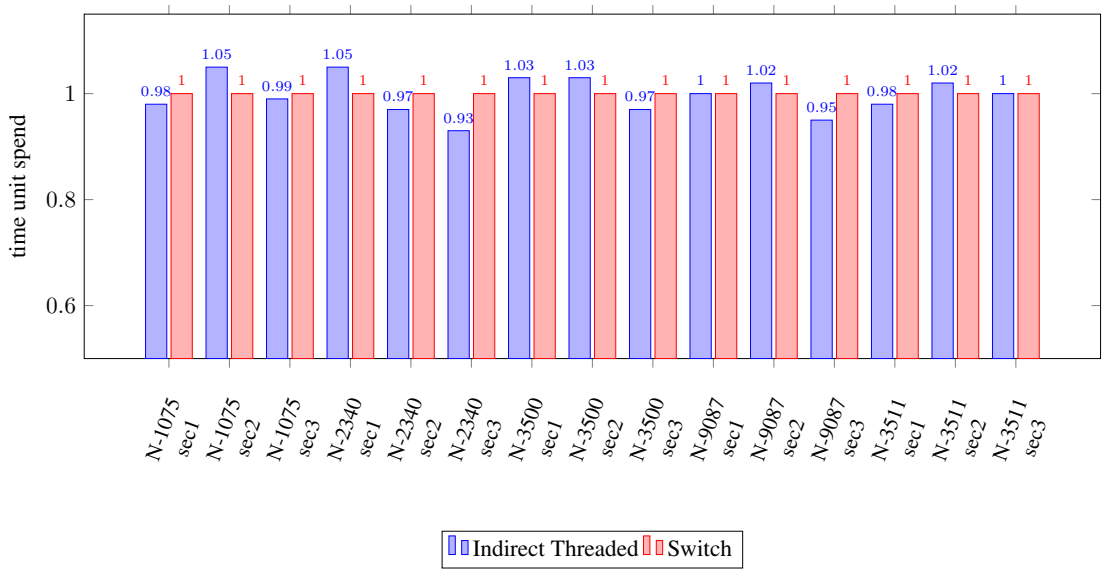


FIGURE 6.7. SVM performance using different dispatch methods

6.2.3 SVM Experiment Result and Evaluation

The implementation of ITC in SVM uses a similar strategy as in the CPython source code: we implement a macro to switch between ITC and switch dispatch freely. We present the result of VPC benchmark here but we also have run SVM with ITC on other benchmarks and get a similar result.

From the result in Figure 6.1, the majority of the benchmarks run equally or slightly faster when using a switch dispatch. For the remaining benchmark, there is only a minor improvement in runtime when executed with ITC, about 1% - 7% faster. We decide not to implement the indirect threaded code as a prioritised feature for several reasons. First, it is because of the moderate performance of ITC based on benchmark results. Second, from the software engineering perspective, ITC requires extra work on portability since ANSI C does not support the label-as-value extension. Finally, threading can confuse a profiling program based on our experience; it makes profiling tool difficult to retrieve its correct function signature when doing the sampling. Soufflé is performance-oriented, and we would like to have an implementation where the profiling results on the performance can be accurate and well understood.

6.3 Performance Model

In this section, we discuss the experiment and evaluation of the performance of the interpreter mode. The goal of the performance model is to give us a more scientific understanding of the components of the overhead in the interpreter, which can be used as a guide for further performance tuning. We first define our performance model, then we discuss the procedure of profiling in detail.

The runtime cost of an interpreter is consists of the following three parts:

- Dispatch cost for each operation.
- Overhead for the extra abstraction between interpreter and data structure.
- The actual computation in the data structure.

The first cost comes from the dispatch loop, and the second cost comes from the extra virtual adapter between the interpreter and the static data structures.

6.3.1 Model Hypothesis and Definition

Let $f_{STI}(s, p)$ (resp. $f_{C++}(s, p)$) be the run time of the STI with query program s and some input p, the cost is measured only for the actual computation, without any other components such as syntax checking, IR optimisations or IO operations.

Let δ_t be the performance difference between STI and synthesised mode for some query program with the same input. The key observation here is that both interpreter and synthesised version of Soufflé share

the same part of code with regarding computation in the data structure. Therefore, we can assume both engines spend equivalent time in the actual computation in the data structure, therefore, the performance difference between two models, $\delta_t = f_{STI}(s,p) - f_{C++}(s,p)$, is reflected by $1)$ the dispatch costs of each operation, $2)$ the extra virtualisation between the interpreter and the data structure adapter.

6.3.2 Calculate Dispatch Cost

There are two general approaches to capture the cost of dispatches in the interpreter mode. We can use an indirect approach by calculating the total running time of the entire program as well as the accumulated runtime of each operations $\sum f_{op_i}$, then the total dispatch time is defined by $f_{dispatch} = f_{STI}(s,p) - \sum f_{op_i}$. Another way is to directly insert a counter between the start and end of each dispatch.

Since STI is executed in a recursive pattern, the profiled runtime of an entire operation also includes the total runtime of all its child-operations. To obtain the dispatch time indirectly, it requires extra data processing after profiling and can be error-prone due to accumulated profiling offset. As a result, we decided to use the direct approach by inserting *PAPI* counter (`PAPI_REAL_CYC`) between the start and the end of each dispatch operation.

As mentioned, PAPI relies on hardware performance counters, which can be challenging to be configured and has unavoidable overhead and accuracy offset, affect the final measurement (Roehl et al., 2014). Especially when measuring short but intensive intervals in the program such as dispatch, the small offset in each measurement can lead into a significant error in the final result (Zaparanuks et al., 2009). The topic of hardware profiling overhead and its impact are beyond the scope of this work. However, we are well aware of the potential offset in our measurement result. The focus on this experiment is not to study and eliminate measurement overhead but rather to provide an overview of the bottleneck components in the STI. Therefore it can be used as a guideline for optimisation. For future work, we are planning to refine the profiling method, for example, by directly accessing hardware counters through rdmpc instruction for lower overhead (Liu and Weaver, 2017; Danalis et al., 2017). For this work, as there is an overhead each time we start/stop the profiling counter, we expect that the measured dispatch time will be more significant than the actual dispatch time.

6.3.3 Calculate Overhead in Data Structure Adapter

The data structure adapter provides a uniformed, virtualised interface for the interpreter during runtime; the trade-off is the virtual call between interpreter and adapter, and the comparator virtualisation which requires tuple to be rearranged before reading and writing. We would like to understand the cost of the virtualisation, by putting the profiling counter (`PAPI_REAL_CYC`) between the interfaces as well as the data rearrangement. Similar to the dispatch overhead, we expect the measurement result to be higher than the real cost because of the profiling overhead in the hardware.

6.3.4 Experiment Result and Evaluation

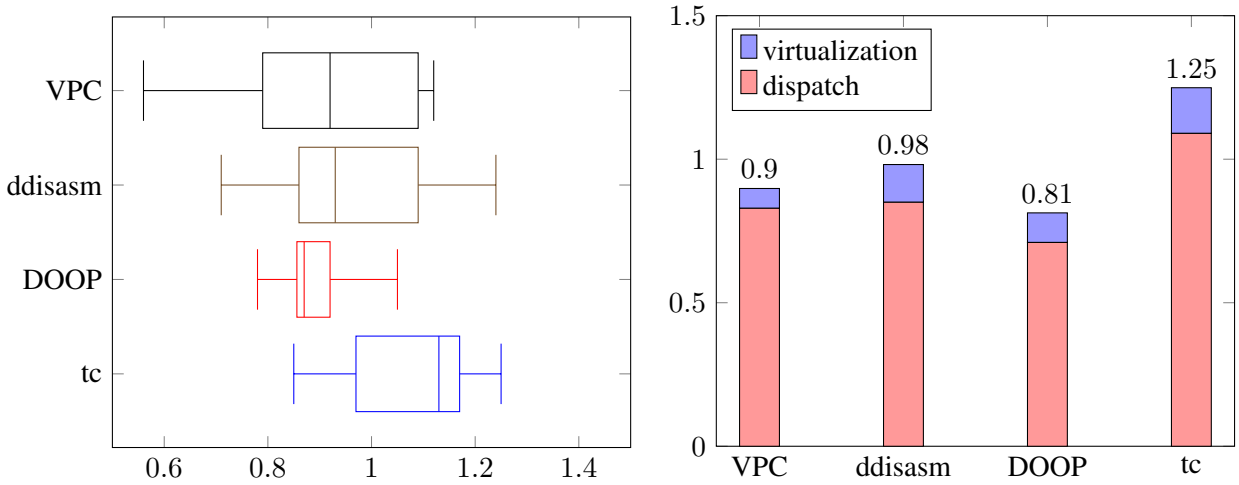
We sum up the measured dispatch time and virtualisation time denoted as $f_{overhead}$; we then compare the result with the actual gap between interpreter runtime and synthesised C++ program. Ideally, $\delta_t = f_{STI}(s,p) - f_{C++}(s,p) = f_{overhead}$. If the measurement capture all the cost components in δ_t, we should be able to cover all the aspects of the runtime overhead and get a result roughly equal to the performance gap. However, taking into account the profiling overhead, we expect that our performance model to have a result greater than the actual runtime gap.

Experiment results are shown in Figure 6.8. Figure 6.8a is a box plot of the coverage of our performance model on each benchmark. Among the four benchmarks, *tc* has the best result where the median of the coverage slightly above 100% and has a relatively compact distribution. All other benchmarks have a midpoint between 0.87 to 0.92 and some extreme outliers, for example, the lower whisker of *VPC* is only 0.56. This is because the *tc* benchmark is a relatively simple program with basic Datalog rules to calculate the transitive closure of the input, it only involves a limit number of RAM operations and is easy for us to investigate those costs in the source program. However, other benchmarks are from the real-world application and are more complicated, involving many auxiliary procedures which are hard to capture. The result is indicating that, although our model can capture the majority of the overhead, it still misses some of the critical performance hot spot, which can be the overhead of runtime variable environment or any auxiliary procedure that is unique to the interpreter.

There are many potential hot spots, but merely measuring and accumulating all of them can lead to a significant error due to overhead in profiling. We are planning to further tunning the profiling method and our performance model in future experiments. In summary, although our performance model can

capture the majority of the overhead, it still missing some of the critical spots when applied to the real-word program.

Figure 6.8b demonstrate the individual coverage of dispatch and virtualization. Although the model can be incompleted, we can still confirm that the virtual dispatch contributes to the majority of the overhead in our performance model. Based on this result, we would like to apply the optimisation technique on the dispatch procedure in order to improve the performance of the interpreter.



(A) Coverage of performance model

(B) Average coverage of dispatch and virtualization

FIGURE 6.8. Performance model evaluation

6.4 Super-Instruction Optimization

In the previous section, we conclude that the dispatching time contributes to the majority overhead of the interpreter mode. In this section, we further investigate the composition of the instruction set on real-word programs and apply appropriate super instruction based on the most frequent executed instruction.

6.4.1 Operation Distribution

For this experiment, we chose the VPC benchmark and analysis the operation distributions in STI. In particular, we would like to know what are the most frequent operations and we would like to apply super-instruction so that we can eliminate the dispatch cost for those operations. Recall that super-instruction is an optimisation techniques targets on the instruction set, by merging two or more virtual instructions into one specialised, extensive virtual instruction.

Figure 6.9 demonstrate the distribution of the instructions. We also compare the distribution input-wise (with a fixed query) and query-wise (with a fixed input). The comparison result suggests the operation distribution does not necessarily strictly determined by the source program itself; for VPC benchmark, the control flow can be data-driven. Although we did not observe this pattern on other benchmarks, the particular characteristic here indicates it is necessary to consider the impact of the input data set when applying specialised optimisation, as they can significantly affect the program control follow.

Back to the operations distribution, we conclude that the most frequent operations are `TupleElement` and `TupleOperation`. `TupleElement` is the operation of fetching an element from the runtime environment, which is used when a computation depends on another previous runtime result. It is always a leaf node in the RAM and shadow tree, hence makes it easy to be integrated into its parent operation. On the other hand, `TupleOperation` is a dummy instruction that has no computational meaning but is used for the profiling part of the interpreter. We decide not to eliminate this operation for this experiment, as the better solution is to ignore this node when generating the shadow tree, depends on whether the profiling option is turned on. For now, we will focus on investigating and optimising `TupleElement`.

6.4.2 Super-Instruction Implementation

Many operation requires result from runtime evaluation, such as `IndexScan` and `Project`. Depends on the use case, the actual expression for evaluating the value can be different, for example, it can be a constant, in which case is represented by a `Number` expression, or it can come from a tuple, in which case is presented by a `TupleElement` expression. In Soufflé, there are generally three cases when evaluating value for computation:

(1) `Number`, in which case, the value to be fetched is known during the compilation the source program and can be determined without runtime context.
(2) `TupleElement`, in which case, the value is the result of some tuple in a relation. The tuple can come from user input or runtime computation, and the exact value cannot be known before runtime. However, the location that stores the element is static and can be known during compilation.
(3) The last case includes all other generic expressions, for which we do not try to make them into a super instruction as it involves many possible targets and can introduce too much complexity into the program.

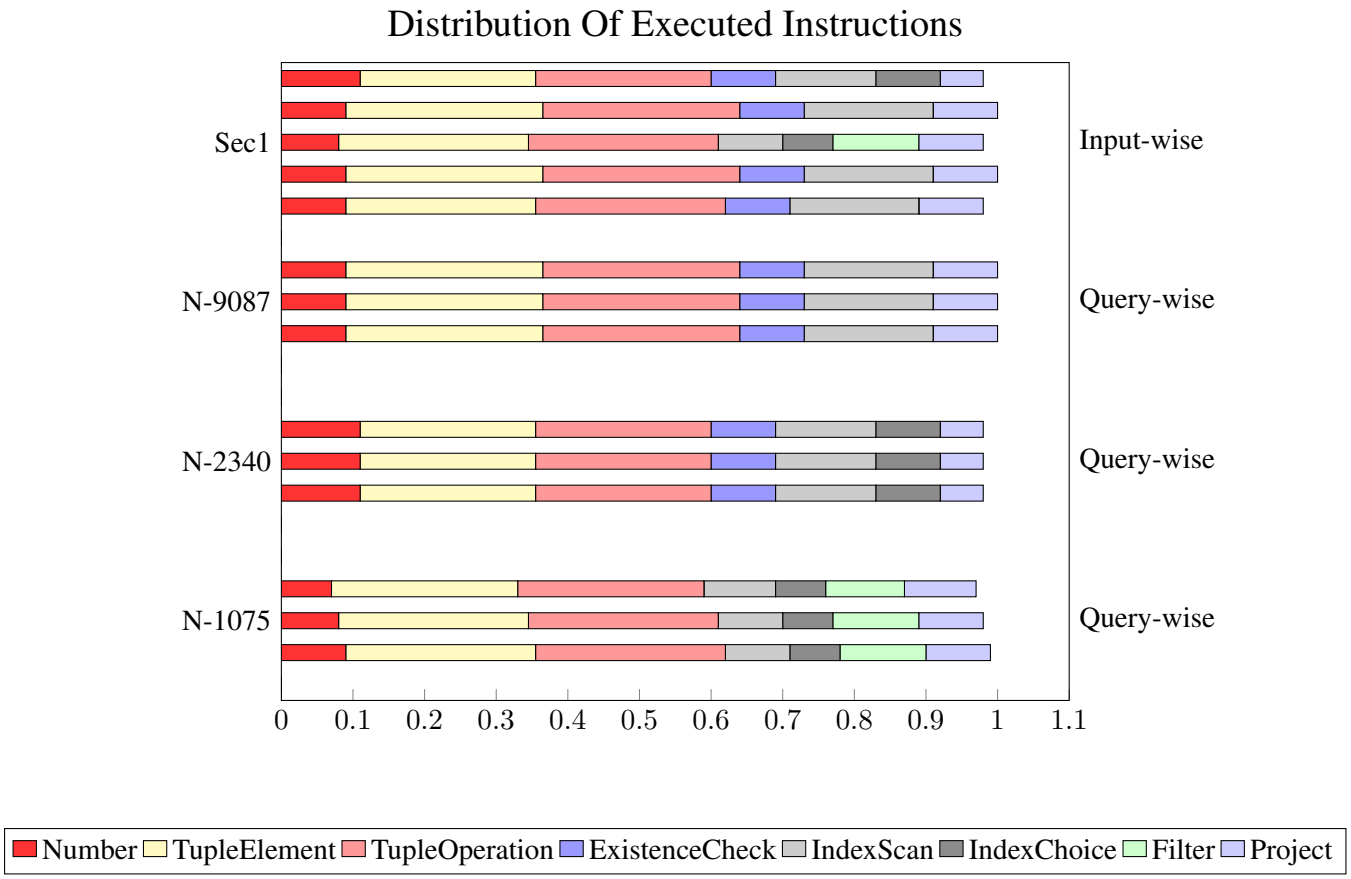


FIGURE 6.9. Instructions distribution input-wise and query-wise. Instructions that are executed less then 1% are discarded.

Since `Number` also contributes to a fair amount of dispatches and since `Number` and `TupleElement` are very similar regarding to the position they can appear in the RAM, we decide to apply super-instruction on both of them. However, a traditional super-instruction approach - where a child is merged with the parent by introducing a new instruction in the instruction set - would not work. Take one possible parent operation `Project` as an example, a `Project` inserts a list of runtime-evaluated value into a relation as a tuple, those values can come from `Number`, `TupleElement`, other generic expressions or a mixture of all. A naive super-instruction cannot work with `Project` since the length of the tuple is not known until runtime and the underlying value expressions can be a mixture, eventually leading to too many possible combinations to build.

To overcome the issue, we introduce three extra fields for possible parent operations in the shadow node. A `constant` filed that stores a list of paired elements, the first index represents the target location to store in the result tuple, and the second index presents the actual number. Similarly, a

`tupleElements` field with a list of paired elements, where the first index 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, define a `GenericExpression` similarly. The code generation example is illustrated in Listing 15.

During execution, we simply evaluate all three fields separately, as shown in Listing 16. Because the field already implies the operation type, there is no need to perform virtual dispatch for `Number` and `TupleElement`. The extra cost here is the data fetching from the extra fields and the non-sequential write operation on the result array, however, we expect that our approach would justify the cost of virtual dispatches and result in a performance gain.

Listing 15 Generating an indexScan operation with super-instruction

```
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;
}
```

6.4.3 Result and Evaluation

The result is shown in Figure 6.2, indicating the reduction in the total number of dispatches and the performance improvement (defined by the ratio between old and new runtime). By eliminating 24.7% dispatches, we achieve a runtime improvement of 1.127 on average. We also observe a similar improvement ratio on other benchmarks. The resulted improvement is slightly less then one would expect based on Amdahl's law, this is because of two reasons; firstly, there is an error offset in the measurement, which does not reflect the actual runtime; secondly, the elimination of the dispatch does not come freely, remember we had to perform non-sequential array access and extra data fetching.

Listing 16 Executing an indexScan operation with super-instruction

```
RamDomain executeProject(ShadowNode* node) {
    /* initialize result tuple */
    std::vector<RamDomain> tuple(n);

    // Evaluate generic expression
    for (auto expr : node->expressions) {
        /* Rely on dispatch to evaluate result */
    }
    // Retrive constant value
    for (auto constant : node->constants) {
        index = constant[0];
        value = constant[1];
        tuple[index] = value;
    }
    // Retrive tupleElement
    for (auto t : node->tupleElements) {
        index = t[0];
        location = t[1];
        tuple[index] = getValueFromRuntime(location);
    }

    /** insert tuple into relation **/
    return result;
}
```

In summary, we perform super-instruction optimisation based on the result of our performance model as well as investigating the operation distribution; the statistics data lead us to eliminate the most frequent and expensive operations and achieve a performance gain as we expect.

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

TABLE 6.2. Performance Result

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this work, we review the interpreter architectures, performance model and optimisation techniques published in recent years. We present a new interpreter architecture - *Switch-based Shadow Tree Interpreter* - a light-weight, tree-walk interpretation technique that also provides good engineering principle and eases the interpreter development. Our own implementation, Soufflé Tree Interpreter, utilises this architecture, resulting in excellent performance and good code quality.

In addition to that, we also implemented Soufflé Virtual Machine - a stack-based Virtual Machine. Although the STI outperforms it, we were able to identify the cause of the slow down, which is the instruction set design. SVM uses a fine-grained instruction set, which cannot work well with the semantic of RAM, causing SVM to spend much more virtual dispatches in the iterative statement. On the other hand, by utilising a coarse-grained instruction set, STI can save 9.91% - 36.36% of dispatches in practice.

We also implemented a data structure adapter to support usage of template class with only runtime information. This is done by forcing the compilation of template class with the factory method and unifying the interfaces with the adapter. We also carefully design the iterator behaviour, amortising the virtual function call by writing data into an internal buffer beforehand. Although the variety of the objects that can be created is limited, we have not met any user case requiring more then we define currently. The adapter enables Soufflé interpreter with parallel execution during runtime and boosts up the performance.

Finally, we experiment with two optimisation techniques targeting on virtual dispatch and instruction set design. We experiment indirect Threaded code on CPython and SVM; we can conclude that ITC still is effective on modern hardware but with much less impact. The result on SVM suggests that it is

currently not worth it to implement ITC in Soufflé as it gives a suboptimal performance and introduces extra complexity into the project. We also build up a naive performance model of STI for performance evaluation. The observation on the performance model suggests that that most of the overhead of STI comes from the dispatch cost. The data structure adapter also contributes a fair amount of overhead, around 10% - 20% based on the benchmark. Further investigating the operation distribution in Soufflé program, we found that 23 - 30% of dispatches come from `TupleOperation` and `Number`. We then perform super-instruction optimisation based on the statistics data, the optimisation manages to get rid of the cost of dispatches but introduce the extra overhead of data inefficiency such as non-sequential access to an array. Overall, our optimisation manages to achieve a speedup around 10% among the benchmarks.

In summary, this project provides a new general strategy to implement tree interpreter. By utilising the strategy with careful implementation, our new interpreter in Soufflé has not only excellent performance but also a maintainable complexity. Compared to the legacy implementation, Soufflé Tree Interpreter can scale well in real-world program and finish in reasonable time. Compared to the synthesiser mode, STI is currently only 2.11 - 5.88 times slower.

7.2 Future Work

7.2.1 Performance Model

As we discussed, the profiling function itself introduces overhead and cannot give an accurate measurement, especially when measuring extreme short but frequent program interval. As a result, the more components we measure, the higher the error in the accumulated final result.

Our performance model is only able to explain part of the performance gap between the synthesiser and the STI when applied to the real-world program. This suggests that we still have uncaptured cost in the program, as well as offset error for the profiling counter. How to better understand the overhead of profiling and the error offset itself is a challenging research problem, but is worth exploring. A good performance model can lead to practical and critical optimisation decision, like what we did in super-instruction optimisation. We are planning to use other measurement methods with lower overhead, such as directly insert rdmpc instruction into the program to access hardware counter during runtime.

More work is required in order to understand hardware profiling better and obtain a nice performance model.

7.2.2 Data Structure Adapter

The data structure adapter provides the interpreter with a unified interface for querying in the data structure and brings the interpreter a significant performance boost. However, it still comes with a price: $1)$ the extra virtual function call between the interface and the interpreter. $2)$ Tuple reordering before inserting and reading. We would like to investigate a better approach to further reduce this overhead with zero-cost virtualisation in C++. The plan is to specialise the virtual instruction set instead, such that each instruction support a particular tuple arity and data structure implementation (template function). As a result, if we are planning to support arity with up to 30 with five potential data structures, it would result in 89 version of different instructions for each instruction currently in RAM. To avoid engineering overhead, we are planning to generate those operations using macro and template. In addition to that, although we conclude that the tuple reordering before *writing* in unavoidable, we can eliminate the reordering when *reading* from the data structure. This can be done by modifying the shadow node during code generation based on the comparator order. For example, when a tuple is inserted with rearrangement $< 0, 2, 1 >$, then in the rest of the execution, all read operation would be modified to cope with this new ordering: a read of second index element in the tuple would be changed to a read of third index element during code generation.

We believe the new strategy can further improve the performance of the interpreter.

7.2.3 Soufflé Tree Interpreter

The tree-like interpreter, although not favoured by many interpreter authors, performs surprising well with Soufflé. This mainly benefits from the SSTI strategy and a straight-forward semantic that fits well in RAM.

However, there is still fundamental performance disadvantage when using a tree-like interpreter; for example, when transferring control to subsequent operations, the interpreter needs to follow a pointer to fetch the child node, which can lead to data inefficiency when compared to a more compacted implementation, such as stack-based implementation. Measuring the exact overhead of the data fetching through the pointer can be extremely challenging as the cost is only reflected in hardware level and often

related to cache events. We would like to investigate further in the STI to understand better about its potential performance.

7.2.4 Soufflé Virtual Machine

The traditional stack-based Virtual Machine instruction does not fit well in the semantic of RAM. Nevertheless, we still believe there is performance potential in VM interpreter if implemented carefully, because a bytecode representation is more data-cache friendly when compared to walking through pointers in the tree interpreter.

We could redesign the iterator-related instruction in order to overcome the issue of low efficiency in the iterative statement. For example, we could give up the pure flat evaluation model and introduce recursion in SVM; this would cause issues on the control of vPC but it would let SVM benefit from the native C++ for-loop support and reduce the total number of dispatches dramatically as in STI. We can also provide a better virtual runtime environment to reduce the overhead of iterator retrieving during execution.

In summary, Virtual Machine based interpreter implementation is still worth exploring. Given we now have a better understanding of the interpreter performance model in Soufflé, we can build a VM that fits better in the RAM semantic.

Bibliography

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

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.

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 2017, page 25–30. Association for Computing Machinery, New York, NY, USA.

John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. 2019. *Reachability Analysis for AWS-Based Networks*, pages 231–241.

David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of c++ virtual function calls. In *In Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 324–341.

James R. Bell. 1973. Threaded code. *Communications of the ACM*, 16(6):370–372.

Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. 2005. Context Threading: A flexible and efficient dispatch technique for virtual machine interpreters **This research was supported by NSERC, IBM CAS and CITO. *International Symposium on Code Generation and Optimization*, pages 15–26.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262.

Stefan Brunthaler. 2011. Interpreter instruction scheduling. In Jens Knoop, editor, *Compiler Construction*, pages 164–178. Springer Berlin Heidelberg, Berlin, Heidelberg.

Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective plus MasteringEngineering with Pearson EText – Access Card Package*. Pearson, third edition.

C. Curley. 1993. Life in the fastforth lane. *Forth Dimensions*, pages 6 – 12.

Anthony Danalis, Heike Jagode, Vince Weaver, Yan Liu, and Jack Dongarra. 2017. New developments for papi 5.6.

Robert B. K. Dewar. 1975. Indirect threaded code. *Communications of the ACM*, 18(6):330–331.

David Ehringer. 2010. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 4(8).

M. Anton Ertl and Gregg David. 2003. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM SIGPLAN Notices*, 38(5):278–288.

M. Anton Ertl and David Gregg. 2001. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, page 403–412. Springer-Verlag, Berlin, Heidelberg.

M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *J. Instruction-Level Parallelism*.

M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. 2002. Vmgen—a generator of efficient virtual machine interpreters. *Software: Practice and Experience*, 32(3):265–294.

Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

Maurizio Gabbrielli and Simone Martini. 2010. *Programming Languages: Principles and Paradigms*. Springer Publishing Company, Incorporated, first edition.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, first edition.

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 '19, page 1176–1186. IEEE Press.

David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. 2005. The case for virtual register machines. *Science of Computer Programming*, 57(3):319–338.

Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On synthesis of program analyzers. In *CAV*.

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019a. 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, page 31–40. Association for Computing Machinery, New York, NY, USA.

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019b. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 327–339. Association for Computing Machinery, New York, NY, USA.

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2020. Specializing parallel data structures for datalog. *Concurrency and Computation: Practice and Experience*.

Anton Korobeynikov. 2007. Improving Switch Lowering for The LLVM Compiler System. In *Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering (SYRCoSE'2007)*. Moscow, Russia.

Richard E. Ladner, James D. Fix, and Anthony LaMarca. 1999. Cache performance analysis of traversals and random accesses. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, page 613–622. Society for Industrial and Applied Mathematics, USA.

Stanley B. Lippman, Jose Lajoie, and Barbara E. Moo. 2012. *C++ Primer*. Addison-Wesley Professional, fifth edition.

Yan Liu and Vincent M Weaver. 2017. Enhancing papi with low-overhead rdpmc reads. In *Programming and Performance Visualization Tools*, pages 3–20. Springer.

Esmira Muslija and Edin Pjanić. 2018. Which container should i use? In *International Symposium on Innovative and Interdisciplinary Applications of Advanced Technologies*, pages 487–504. Springer.

Patrick Nappa, David Zhao, Pavle Subotic, and Bernhard Scholz. 2019. Fast parallel equivalence relations in a datalog compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 82–96. IEEE Computer Society, Los Alamitos, CA, USA.

Ian Piumarta and Fabio Riccardi. 1998. Optimizing direct threaded code by selective inlining. *ACM SIGPLAN Notices*, 33(5):291–300.

Thomas Roehl, Jan Treibig, Georg Hager, and Gerhard Wellein. 2014. Overhead analysis of performance counter measurements. *2014 43rd International Conference on Parallel Processing Workshops*, pages 176–185.

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

Martin Schoeberl. 2005. Design and implementation of an efficient stack machine. volume 2005.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206. Association for Computing Machinery, New York, NY, USA.

Michael L. Scott. 2009. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, third edition.

Jiwon Seo, Stephen Guo, and Monica S Lam. 2013. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE.

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. 2008. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4).

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149.

Lagouvardos Sifis, Dolby Julian, Grech Neville, Antoniadis Anastasios, and Smaragdakis Yannis. 2020. Static analysis of shape in tensorflow programs. In *The European Conference on Object-Oriented Programming*, pages 15:1 –15:30.

Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Victor Stinner. 2020. Pyperformance. `https://github.com/python/pyperformance`.

Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153.

Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lindholm Tim, Yellin Frank, Bracha Gilad, Buckley Alex, and Smith Daniel. 2020. *The Java Virtual Machine Specification*. 14. Oracle America, 500 Oracle Parkway, Redwood City, California 94065, U.S.A., 20th edition. An optional note.

Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer.

Clark Wiedmann. 1983. A performance comparison between an apl interpreter and compiler. In *Proceedings of the International Conference on APL*, APL '83, page 211–217. Association for Computing Machinery, New York, NY, USA.

Phil Winterbottom and Rob Pike. 1997. The design of the inferno virtual machine. In *in In IEEE Compcon*, pages 241–244.

D. Zaparanuks, M. Jovic, and M. Hauswirth. 2009. Accuracy of performance counter measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32.

David Zhao, Pavle Subotic, and Bernhard Scholz. 2020. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Transactions on Programming Languages and Systems*, 42:1–35.

.1 Details on RAM

The `RamNode` is the basic class of the RAM tree node. The overview of class hierarchy is shown in Figure .1. `RamRelation` represents the relation in the RAM IR. `RamProgram` is the main entry of the program, it includes the root of the tree as well as relation declarations and subroutines.

.1.1 RAM

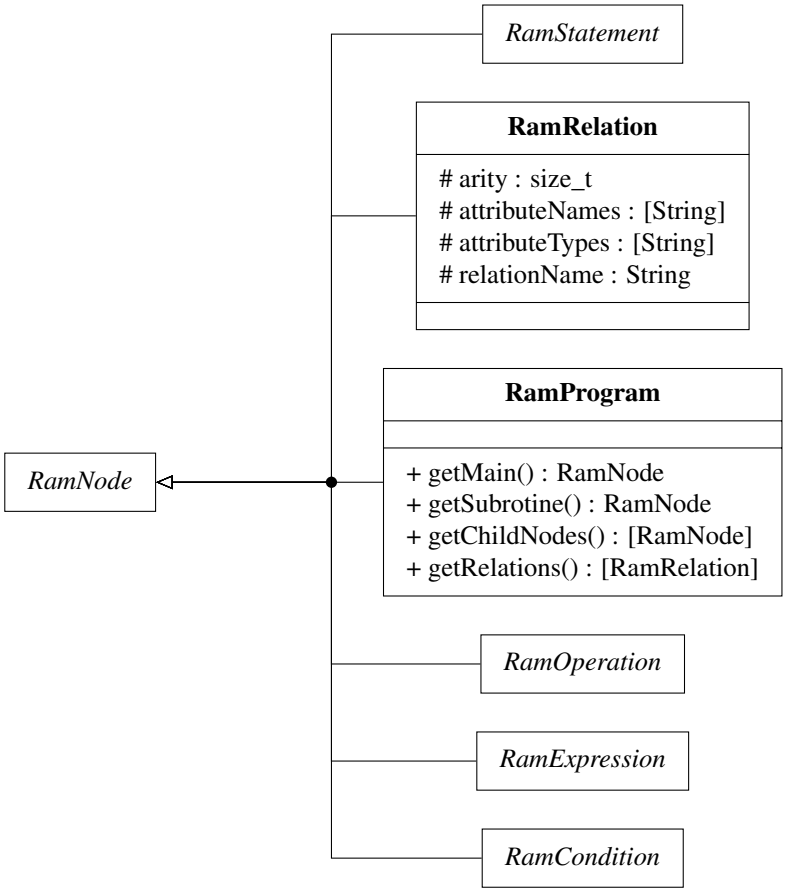


FIGURE .1. RAM

.1.2 RAM Statement

The `RAMStatement` class hierarchy is shown in Figure .2.

- `RamBinRelationStatement` describes a statement that operates on two relations. `RamExtend` statement extends the content of the first relation into the second one. `RamSwap` statement

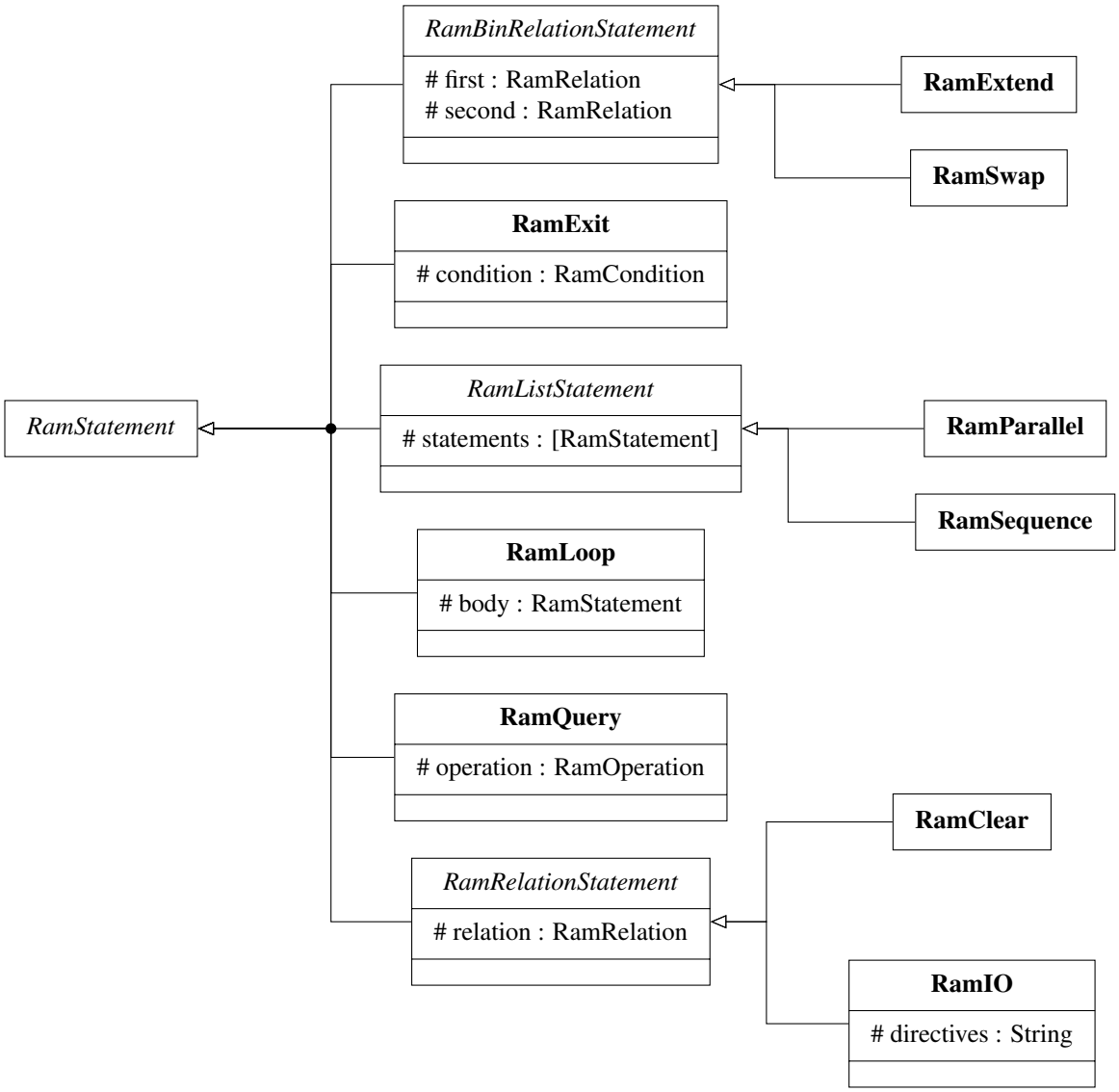


FIGURE .2. RAM Statement

swaps the content (tuples) of two relations. Both statement requires the two argument relations be of the same type.

- `RamListStatement` represents a list of Ram statements. They can be either executed in sequential (`RamSequence`) or in parallel (`RamParallel`) depends on user specification.
- `RamLoop` statement is used with one or more `RamExit` statement. It computes a fixed-point evaluation until one of the condition in `RamExit` is met.
- `RamQuery` statement corresponding to a core machinery of the semi-native evaluation.
- `RamClear` statement cleans up the content of a given relation.

- `RamIO` statement describes the IO operations of the target relation. It handles input/output of a relation content.

.1.3 RAM Operation

The `RAM Operation` class hierarchy is shown in Figure .3.

- `RamAbstractConditional` describes the conditional branch operations. `RamFilter` executes the nested operation only if the result of its condition returns true.A `RamBreak` would break the current program control out of the nested operations if the condition holds.
- `RamProject` projects the result of the expressions into target relation.
- `RamScan` iterates all the tuple in the target relation.
- `RamChoice` iterates all the tuple in the target relation, stop and return the tuple if the condition holds.
- `RamAggregate` performs aggregation on the given relation, e.g. summation, find maximum/minimum value.
- `RamIndexOperation` describes the indexed version of basic RAM operations, it only iterates the tuples follows a certain search pattern in the given relation.

.1.4 RAM Expression

The `RAM Expression` class hierarchy is shown in Figure .4.

- `RamTupleElement` access and return element from the current tuple in the runtime environment.
- `RamConstant` returns a constant value.
- `RamPackRecord` evaluates and store a user-defined record.
- `RamSubroutineArgument` access and return the n^{th} argument of a subroutine.
- `RamAbstractOperator` is the abstract class of an operator/function. `RamUserDefinedOperator` represents an extrinsic (user-defined) functor. `RamIntrinsicOperator` represents an intrinsic (built-in) functor.

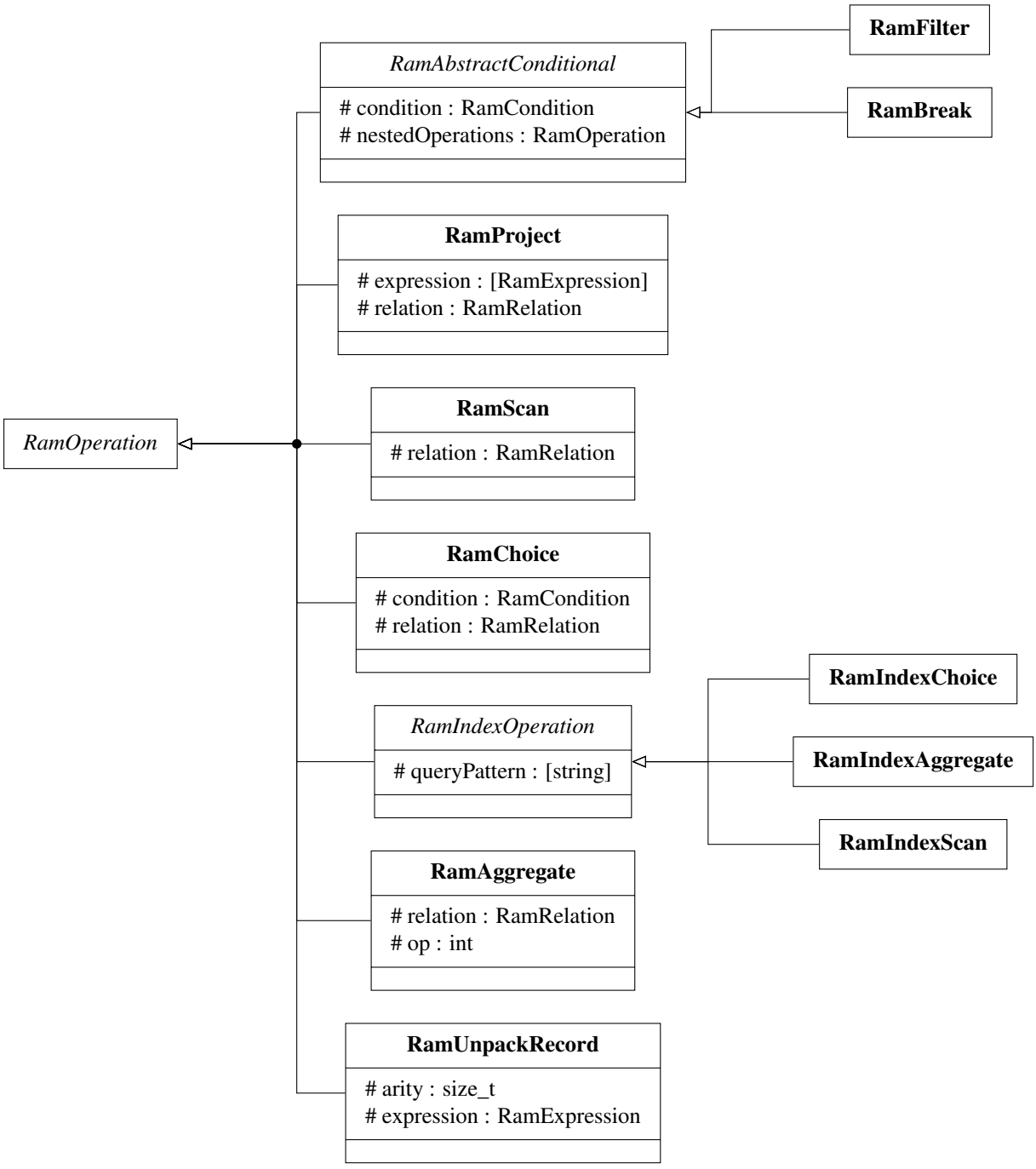


FIGURE .3. RAM Operation

.1.5 RAM Condition

The `RAM Condition` class hierarchy is shown in Figure .5.

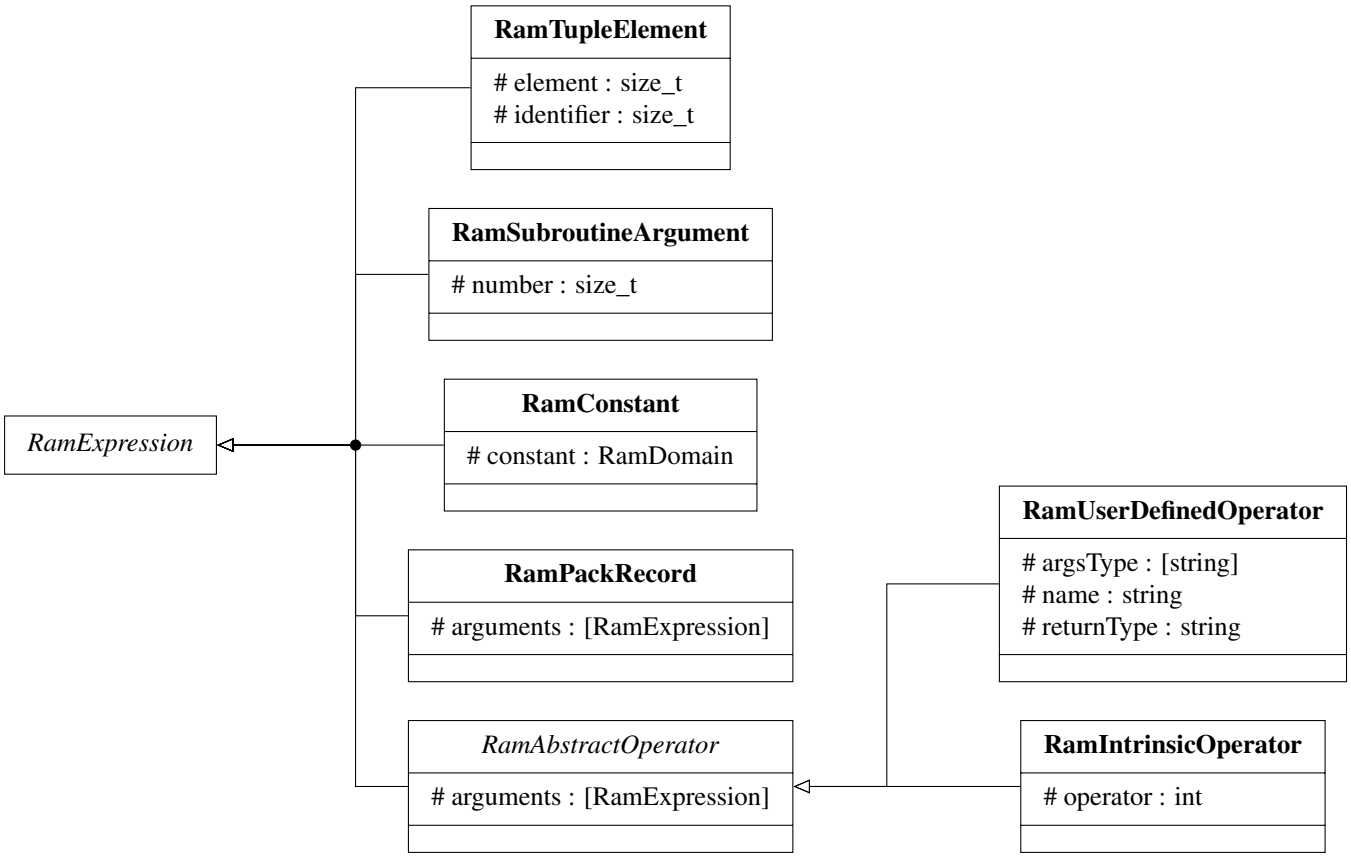


FIGURE .4. RAM Expression

- `RamNegation` negates the result of evaluating a `RamCondition`.
- `RamEmptinessCheck` returns true if the given relation is empty.
- `RamExistenceCheck` computes the tuple value by evaluate the expressions and returns true if the given relations contains tuple with the same value.
- `RamConjunction` returns the conjunction results of left-hand-side and right-hand-side expressions.
- `RamConstraint` represents a binary constraint operation, e.g. less-equal.

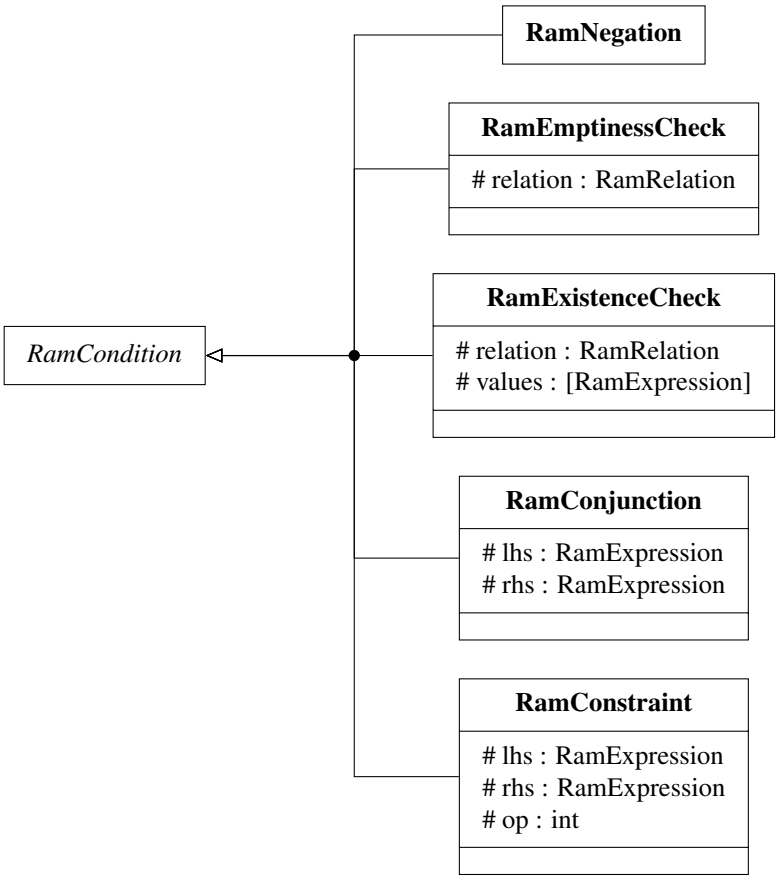


FIGURE .5. RAM Condition