Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Optimization of a Symbolic-Execution-Based Program Verifier

Bachelor's Thesis

Moritz Knüsel

September 13, 2019

Advisors: Prof. Peter Müller, Dr. Malte Schwerhoff
Department of Computer Science, ETH Zürich

Abstract

This thesis presents an investigation into performance problems exhibited by Silicon, a backend verifier for the Viper verification infrastructure. We identify possible reasons for the slow performance of several example files, and, for some, present possible improvements.

These improvements are implemented in Silicon and their performance impact assessed. Not all of them turn out to be worth the additional code complexity in the end. We also make the case for changing how Silicon communicates with the SMT-Solver it relies on, by carrying out a performance comparison with another verifier that supports both this other mode of operation and the one Silicon currently uses.

Contents

Contents **iii**

1 Introduction **1**

2 Background **3**
- 2.1 Viper . . . 3
  - 2.1.1 The Viper Language . . . 4
- 2.2 Symbolic Execution . . . 5
- 2.3 Satisfiability Modulo Theories . . . 6
  - 2.3.1 E-Matching . . . 6

3 Example Collection & Benchmarking **9**
- 3.1 Profiling . . . 9

4 Examining Slow Examples **11**
- 4.1 Nested Disjunctions . . . 11
  - 4.1.1 The Example Program . . . 11
  - 4.1.2 Performance . . . 11
  - 4.1.3 How Silicon Handles Short-Circuiting . . . 13
  - 4.1.4 A Different Approach . . . 14
- 4.2 Partial Snapshot Maps . . . 14
  - 4.2.1 Background . . . 14
  - 4.2.2 The Necessity of the Equality Axiom . . . 17
  - 4.2.3 An Ordering on PSMs . . . 17
  - 4.2.4 Dynamically Tightening Triggers . . . 19
- 4.3 Weak Memory Verification . . . 21
  - 4.3.1 What's Happening . . . 23
  - 4.3.2 The Matching Loops . . . 23
  - 4.3.3 Trying to avoid the Matching Loops . . . 24
  - 4.3.4 Conclusion . . . 25

5 Solver Interface **27**
5.1 Measurement 28
5.2 Results 28

6 Resource Bounds **31**
6.1 Background 31
6.2 Problems with Timeouts 31
6.3 Implementing Resource Limits 32

7 Miscellaneous **33**
7.1 Z3's Parallel Mode 33
7.2 Bug Fixes 33
7.2.1 Silicon 33
7.2.2 VeriFast 33
7.2.3 AxiomProfiler 34

8 Future Work **35**

Bibliography **39**

Chapter 1

Introduction

Automated Software Verification is a field that has made a lot progress in recent times. Being able to prove programs correct is more important than ever, and continuing research allows us to reason about ever more complex programs. Viper is a language and a set of tools developed at ETH [10], providing an architecture on which new verification tools and prototypes can be developed simply and quickly. Silicon [11] is one of two backend verifiers for the Viper project, and is based on symbolic execution.

The Viper intermediate language is the target of a variety of front-end tools, and is also used to write programs directly. However, users of Viper may sometimes be confronted with unusually long verification times. It is often not clear why some inputs take longer than others, and how to ameliorate the situation.

The goal of this thesis is to investigate performance problems exhibited by Silicon, and improve the performance if possible.

Chapter 2

Background

2.1 Viper

The Viper infrastructure is comprised of an intermediate verification language, which is also called Viper, automatic verifiers for the intermediate language, and a number of front-end tools. Figure 2.1 shows an overview of the Viper infrastructure.

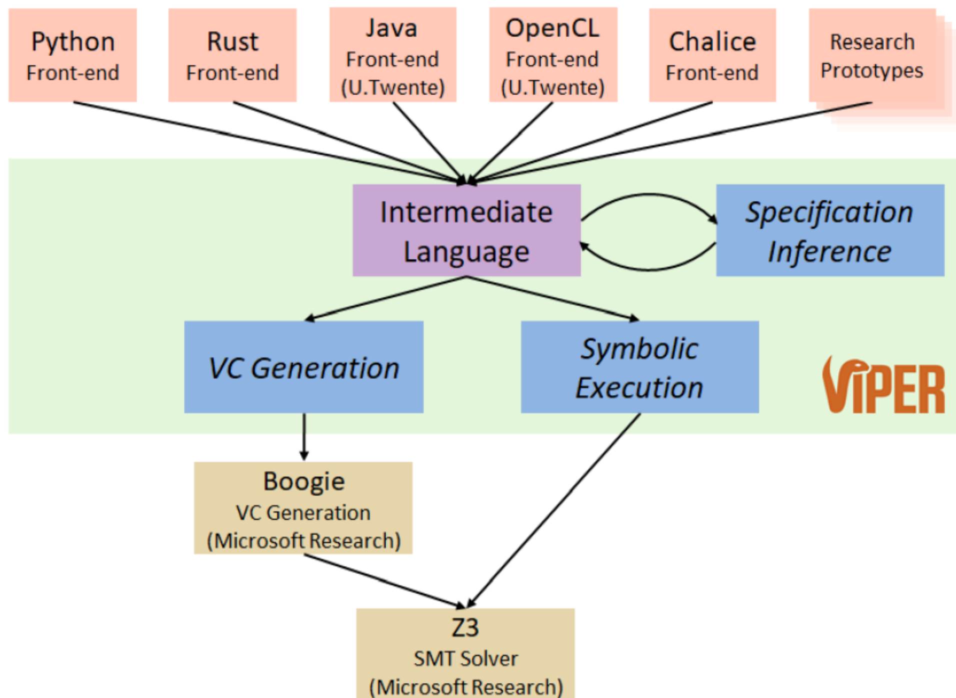

Figure 2.1: An Overview of the Viper Infrastructure

Viper currently has two backend verifiers. Carbon is based on verification condition generation, whereas Silicon is based on symbolic execution. This thesis is only concerned with Silicon.

2.1.1 The Viper Language

See [14] for a general introduction to Viper. We'll quickly go over the features that are most important for this thesis.

Permissions

Viper reasons about the heap using *field permissions*. If a statement, expression or assertion accesses a field, it needs to hold the appropriate permission to that field. Access to a field can be asserted using the accessibility predicate *acc*. Consider the following Viper program:

```
field f: Int
method m(x: Ref)
  requires acc(x.f)
{
        x.f := x.f + 1
}
```

We declare a field `f` and a method `m`. `m` takes a reference `x` as an argument, and wants to modify the heap location `x.f`. To do this, it needs access to `x.f`, which is given by the precondition. Omitting the precondition would make this example not verify, as `m` would not have the necessary permissions to modify `x.f`.

We can also specify *quantified permissions*. In the following example, we require access to the field `f` of every reference in a set.

```
field f: Int
method m(xs: Set[Ref], x:Ref)
  requires forall n: Ref :: n in xs ==> acc(n.f)
{
        if(x in xs) {
                x.f := x.f + 1
        }
}
```

When we write a quantification, we can also specify what triggers should be used, by putting it in curly braces like so:

```
forall n: Ref :: {n.f} n in xs ==> acc(n.f)
```

Functions

Viper allows us to specify functions. Functions in Viper are side-effect free, meaning they cannot affect the program state. A function definition looks like this:

```
function f(...): T
  requires A
  ensures  E1
{ E2 }
```

It defines a function f with a possibly empty list of arguments, returning a type T. The precondition, indicated by the `requires` keyword, is an assertion, whereas the postcondition, indicated by `ensures`, must be an expression. Preconditions therefore allow resource assertion, such as requiring access to a certain heap location, while postconditions do not. The body of the function is optional. If it is omitted, we have an abstract function. Abstract functions are useful when we only care about the function's pre- and postconditions, and not so much about how it actually works.

2.2 Symbolic Execution

Symbolic execution is a technique used to test, debug and verify programs. Whereas under normal execution, the program would manipulate concrete values, under symbolic execution the program manipulates symbolic values. For instance, a statement such as `x = y + 1`, when executed concretely in a state where x and y are, say, 5, would produce a new concrete state where x is now 6. Under symbolic execution, we might instead assign the symbolic values `X` and `Y` to x and y. Executing the statement symbolically in that state, we would obtain a new state which now records that x has the symbolic value `Y + 1`.

One consequence of executing with symbolic values is that we might encounter a branch in the program where it is not known whether the branch will be taken or not. In that case we can execute both branches, one under the assumption that the branching condition is true, and the other under the assumption that the condition is false. For example, if we encountered a branch that depended on a certain variable to be an even number, we would execute one branch assuming it is indeed even, and the other assuming it was instead odd.

During symbolic execution, the execution engine frequently needs to know if certain propositions involving the symbolic values are true or not, for instance if it needs to determine if an assertion holds or if a branch is known to be taken or not. Many symbolic execution engines deal with this by employing an handing these propositions off to an SMT-Solver, a program

```
(declare-var a Int)
(declare-var b Int)
(assert (= a (+ b b)))
(assert (not (= a 0)))
(check-sat)
```

Figure 2.2: A simple SMT-LIB script

intended to solve instances of a problem known as Satisfiability Modulo Theories. In Silicon's case, the SMT-solver used is Z3 [4].

2.3 Satisfiability Modulo Theories

The Satisfiability Modulo Theories problem, SMT for short, is the problem of determining if a formula in first-order logic is satisfiable, with respect to some theory. In pure first order logic, functions and constant symbols are uninterpreted. A theory adds constrains to certain functions and predicates, such as the theory of arithmetic, in which function $+$ has a fixed interpretation in that it adds two numbers together. These theories allow the solver to use specialized inference methods for each supported theory, which is more efficient in practice compared to general-purpose theorem proving. Many SMT-solvers support the use of SMT-LIB, a language for specifying SMT problems [1]. The SMT-LIB language is based on S-expressions, which are a simple representation of an abstract syntax tree first used in the Lisp programming language. An S-expression is either an atom, such as `a` or `1`, or a list of S-expressions enclosed by parentheses, such as `(a b c)`. An application of a function `f` to some argument `a` is written as the list `(f a)`. S-expressions were chosen because they are easy to parse, and not primarily for human readability. This is justified by the fact that most SMT-LIB code is not written by humans but generated by automated tools. In SMT-LIB, terms are sorted, which means every term has a type. For instance, the term $(+11)$ has the type Integer. SMT-LIB does not distinguish terms and formulas syntactically, formulas are just terms with boolean type.

A small SMT-LIB script is shown in figure 2.2. This script asks the solver to check if the formula $a = b + b \wedge a \neq 0$ is satisfiable.

2.3.1 E-Matching

E-Matching is one approach to solving the satisfiability of quantified formulas such as $\forall x.P(x)$ [9]. E-Matching is generally more useful in finding out that a certain formula is unsatisfiable. Typically, during proof search the solver will try and generate instances of the quantified terms, which will hopefully be the instances needed to prove unsatisfiability. Consider for example the following formula, where P is an uninterpreted predicate and f

an uninterpreted function.

$$P(f(1)) \wedge \forall x.(P(f(x)) \implies x = 0)$$

If the solver decided to instantiate the quantifier with $x = 1$, it would learn that $P(f(1)) \implies 1 = 0$, which, together with the already known fact that $P(f(1))$ is true, leads to the contradiction that $1 = 0$, and it could successfully prove the formula unsatisfiable.

It is generally not easy to figure out which instances are going to be useful. One approach is to use triggers. A trigger for a quantified formula is a term containing the quantified variables. The quantified term is only instantiated if a term that matches the trigger is already known. For the above formula, we might choose $P(f(x))$ as a trigger. The fact $P(f(1))$ matches this trigger, and would lead to the quantification being instantiated with $x = 1$, leading to the desired contradiction.

Triggers need to be matched with all current knowledge about equalities in mind, simply using syntactic equality is often not enough. Consider the following modification to the above formula:

$$P(a) \wedge a = f(1) \wedge \forall x.(P(f(x)) \implies x = 0)$$

Looking for a term that syntactically matches the trigger $P(f(x))$ would not lead to a match. We could choose $f(x)$ as a trigger, but that might lead to a lot of unhelpful instantiations if we had other terms involving f that are not immediately related to the assertion that $\forall x.(P(f(x)) \implies x = 0)$. Instead, we match with respect to the equalities asserted in the current context. In our example, substituting 1 for x would make $P(a)$ equal to $P(f(x))$.

In some cases, it's not possible to come up with a single trigger that contains all the quantified variables. In that case, we can use multitriggers. A multitrigger is a set of triggers, that together contain all quantified variables, and have to match simultaneously. It is also possible to specify alternative triggers, where a match for any of the triggers leads to a new instantiation.

The problem that remains is how to choose the right triggers. We have to strike a balance between triggers that are too restrictive and triggers that are too liberal. If we make our trigger too restrictive, we might miss relevant instantiations. Consider the following formula:

$$\neg(a = b) \wedge f(a) = f(b) \wedge \forall x.(g(f(x)) = x)$$

The quantification assert that g is the inverse of f. The formula is not satisfiable. If $f(a)$ and $f(b)$ are equal, so are $g(f(a))$ and $g(f(b))$. From the quantification, we can infer that $g(f(a)) = a$ and $g(f(b)) = b$. We can conclude that $a = b$, which contradicts the asserted fact that $\neg(a = b)$. If we

choose $g(f(x))$ as the trigger for the quantification, it won't get instantiated as none of the other terms match the trigger. Thus the solver never learns that $g(f(a)) = a$ nor that $g(f(b)) = b$ and will not be able to prove the formula unsatisfiable.

On the other hand, if we choose a trigger that's too liberal, we can get a lot of unnecessary instantiations. It might even happen that a term we learn from the instantiation again matches the trigger, leading to a new instantiation which again matches the trigger, and so on and so forth. Consider $\forall x.(\neg(g(f(x)) = g(x))$. Say we choose $g(x)$ as a trigger. Any other term $g(a)$ would match it, leading to the instantiation $\neg(g(f(a)) = g(a))$. From this we get the new term $g(f(a))$, which again matches the trigger $g(x)$. Such a loop is known as a matching loop.

SMT-LIB supports user-defined triggers under the name patterns. If none is given, the solver chooses the triggers on its own.

Chapter 3

Example Collection & Benchmarking

Examples were collected mainly by reaching out to developers of frontend tools that use the Viper infrastructure. Some of the slower cases from Silicon's own testsuite were also included. All in all, about 80 examples were collected.

For benchmarking, we used the Viper-Runner tool, with some modifications. Viper-Runner sets up a Nailgun server, which maintains a JVM and allows clients to run programs on that JVM. This removes the JVM startup overhead from the measurements. Viper-Runner then runs Silicon on a number of predefined files and records some statistics such as execution time and exit code. The modifications to Viper-Runner mainly enabled more customizability of how exactly a benchmarking session would proceed. The collected examples have been stored in a bitbucket repository that was shared among students working on Silicon [3].

3.1 Profiling

For profiling, Silicon was equipped with eBPF tracepoints. eBPF stands for extended Berkeley Packet Filter, and is a tracing infrastructure for Linux [8]. eBPF evolved out of the original Berkeley Packet Filter, and was originally used for capturing and filtering network packets, where filters were implemented as programs to run on a virtual machine inside the kernel. eBPF was introduced with a new virtual machine to make better use of modern hardware when the design of the original BPF virtual machine started to become outdated.

eBPF allows us to attach programs to tracepoints, and have these programs run whenever the associated tracepoint is triggered. As an example, a program attached to a memory allocation tracepoint could record a histogram of allocation sizes to help understand the allocation behaviour of an appli-

cation. The kernel itself defines a variety of tracepoints, but user programs can also define tracepoints. User level tracepoints are called User Statically-Defined Tracing (USDT) probes. For Silicon, a number of USDT probes were created and put in a shared library, where they can be triggered from within Silicon.

The eBPF programs for this project were written using bpftrace [5], a high level language for writing EBPF programs. We modified bpftrace to provide its output in the form of an S-expression representing an associative list, for easier use by subsequent scripts.

We chose eBPF for a number of reasons. First, bpftrace has some nice features for data collecting, such as builtin support for recording the average, total and maximum of a value, and builtin histograms. Second, using EBPF, we can use the same tools for collecting data from Silicon's side as well as from a number of other sources. For instance, the JVM itself defines a number of USDT probes, such as garbage collection or object creation probes.

However, in hindsight the other sources of tracepoints such as the kernel or the JVM turned out to be of minor interest, and using bpftrace slightly complicated the benchmarking. eBPF programs are also limited in what they can do, which is necessitated by the fact that the run in the kernel itself. To load an eBPF program in the kernel, it has to pass a number of checks by the in-kernel verifier, such as a termination check and out-of-bounds checks. This, together with the limited stack space available, means we often have to work around problems with limited resources and program complexity. It may have been simpler to just implement the data collection in Silicon directly.

Chapter 4

Examining Slow Examples

4.1 Nested Disjunctions

4.1.1 The Example Program

The first example we look at is a constructed one, that is it was built specifically to expose a performance issue. The program consists of a predicate with a disjunction, and a method that repeatedly uses the predicate. The program can be parametrited by the number N of non-constant operands to the disjunction. Furthermore, the disjunction can be parametrized by how the expression is paranthesised. We call one nesting order the "slow order", with the other one being the "fast order". Figure 4.1 is a listing of the program, with $N = 5$ disjunct terms in the fast order. Figure 4.2 shows the slow version of the predicate for $N = 5$.

By default, Silicon does short-circuit evaluation for logical And (`&&`), Or (`||`) and Implication (`==>`). Short-circuiting means that operands to logical connectives are evaluated in order, but only as long as the value of the whole expression is not determined. For example, when evaluating an expression like `E1 || E2`, where `E1` and `E2` may be arbitrary expressions, `E1` is evaluated first, and `E2` is only evaluated if `E1` evaluates to `false`. If `E1` evaluates to `true`, the value of `E2` doesn't matter, since `(E1 || E2) = (true || E2) = true`. Thus `E2` doesn't need to be evaluated at all. In order to model this in symbolic execution, we evaluate `E1` as normal, obtaining a symbolic value `e1`, and then branch on the value of `e1`, where one path evaluates `E2` under the additional constraint that `e1` is false, and the other path continues without evaluating `E2`, under the constraint that `e1` is true on that path.

4.1.2 Performance

Verification times for both orders are given in Figure 4.3, which should also make clear why the orders were dubbed "fast" and "slow". Using the slow

```
field val: Int

predicate Slow(this: Ref) {
    acc(this.val) && ((this.val >= 0 && this.val <= 4) ==>
        (this.val == 1 || (this.val == 2 ||
        (this.val == 3 || (this.val == 4 ||
        this.val == 0)))))
}

method havoc() returns (res:Int)

method test(this: Ref)
    requires Slow(this)
    ensures Slow(this)
{
    unfold Slow(this)
    var tmp: Int
    tmp := havoc()
    this.val := tmp
    fold Slow(this)
    // repeat
}
```

Figure 4.1: The program for $N = 5$, using the fast order

```
predicate Slow(this: Ref) {
    acc(this.val) && ((this.val >= 0 && this.val <= 4) ==>
        (((((this.val == 0 || this.val == 1)
        || this.val == 2) || this.val == 3)
        || this.val == 4))
}
```

Figure 4.2: Only the predicate for $N = 5$, using the slow order

	Verification Time		Rendering Time	
N	Fast Order	Slow Order	Fast Order	Slow Order
4	2.3s	2.8s	0.13s	0.41s
5	2.5s	3.5s	0.16s	0.98s
6	2.8s	6.0s	0.18s	2.8s
7	3.0s	13.2s	0.21s	8.5s
8	3.1s	33.2s	0.24s	24.3s
9	3.2s	99s	0.24s	72.1s
10	3.7s	290s	0.26s	221s

Figure 4.3: Verification times and rendering times for different versions of the program for both slow and fast order

order makes the verification time increase exponentially. Under the heading "Rendering Time", the table also shows the time spent translating terms from their representation inside Silicon to corresponding SMTLib expressions.

4.1.3 How Silicon Handles Short-Circuiting

The problem with this program is the way Silicon handles short-circuiting evaluation of `||` and `&&` operations.

Silicon evaluates an expression `E1 && E2` by first evaluating `E1`, resulting in a term `e1`. It then creates a new local variable `v`. With that, it evaluates `v ==> E2` in a state where the value of `v` is `e1`. Evaluating an implication short-circuits if the first operand is false, so the short-circuiting behaviour of `&&` is achieved. The local variable was introduced to avoid evaluating `E1` twice. Silicon evaluates the implication to the term `(=> e1 e2)`, and constructs the term `(and e1 (=> e1 e2))` for `E1 && E2`.

Evaluation of `E1 || E2` works similarly. First `E1` is evaluated, a fresh variable `v` with value `e1` is introduced, then `!v && E2` is evaluated. This allows us to utilize the short-circuiting behaviour of `&&` to implement the short-circuiting of `||`. Ultimately, Silicon generates the term

```
(or e1 (and (not e1) (=> (not e1) e2)))
```

for the expression `E1 || E2`.

This expansion can lead to large terms if `e1` itself is large. If we start evaluating disjunctions where the first operand is itself a disjunction, as found in figure 4.2, the size of the term grows exponentially with the number of operands.

	Verification Time		Rendering Time	
N	Fast Order	Slow Order	Fast Order	Slow Order
4	2.1s	2.6s	0.14s	0.16s
5	2.3s	2.1s	0.13s	0.13s
6	2.1s	2.1s	0.14s	0.12s
7	2.1s	2.3s	0.13s	0.14s
8	2.3s	2.6s	0.16s	0.16s
9	2.6s	2.6s	0.18s	0.17s
10	2.7s	2.7s	0.19s	0.19s

Figure 4.4: Verification and rendering times for the new approach

4.1.4 A Different Approach

Our approach works as follows: To evaluate an expression such as `E1 || E2`, we first decompose it into it's subexpression, obtaining a list `[E1, E2]`. If any of `E1` or `E2` is itself an `||`-expression, it is also decomposed. This is repeated until the list doesn't contain any `||`-expressions. Thus we end up with a list of n expressions $\mathtt{E}_i$, where $0 \leq i < n$. These expressions are evaluated in order, starting at $\mathtt{E}_0$. $\mathtt{E}_0$ is evaluated to form a term $\mathtt{e}_0$. Execution branches on $\mathtt{e}_0$. If $\mathtt{e}_0$ is false, we continue evaluating the list, and if it is true, we stop evaluating. The final term is formed as the disjunction of the individual $\mathtt{e}_i$.

The evaluation of `&&` works analogously.

The timings for this approach are shown in figure 4.4. The decomposition of nested operations into a flat list completely removes the differences between the slow and fast orders. As a side effect, a testcase in Silicon's testsuite, which Silicon previously failed to verify, now verifies successfully.

This change had little effect on any other files, since very deeply nested conjunctions or disjunctions don't really occur in regular programs.

4.2 Partial Snapshot Maps

4.2.1 Background

Encoding of Heap-Dependent Functions

Functions in Viper generally may depend not only on their inputs, but also on the values of locations in the heap. Such heap-dependent functions are encoded to the solver by adding an additional argument to the function, called a snapshot, which represents the symbolic values of heap locations that the function depends on.

Snapshots are encoded for the solver as follows: First we define the sort `Snap` for use as the sort of snapshots. We then define `unit`, which is the empty snapshot, as well as the functions `pair`, `first` and `second`. `pair` combines two snapshots to form a new one, and `first` and `second` are used to deconstruct a pair into its components. We also add a number of functions named $\texttt{box}_S$ and $\texttt{unbox}_S$, parametrised by a sort S. These are used to embed values of sort S into `Snap`, using $\texttt{box}_S$, and to get them back out again using $\texttt{unbox}_S$.

Let's examine how heap-dependent functions are encoded using the following example:

```
field f: Int
field g: Bool
function fun(x: Ref): Int
  requires acc(x.f)
  requires acc(x.g)
```

The solver receives the following declaration:

```
(declare-fun fun (Snap Ref) Int)
```

Say `fun` is applied to an argument `x` of sort `Ref`, and the symbolic values of the heap locations `x.f` and `x.g` are `i` and `b`, respectively. Silicon encodes the application like this:

$\texttt{(fun (pair (box}_{\text{Int}}\texttt{ i) (box}_{\text{Bool}}\texttt{ b)) x)}$

Notice that the structure of the snapshot follows from the precondition of the function. Had we put the two `requires` clauses in reverse order, the components of the snapshot pair would likewise be reversed.

Quantified Permissions

Consider the following abstract function:

```
field next: Ref
function fun(refs:Set[Ref]): Set[Int]
  requires forall n:Ref :: n in refs ==> acc(n.next)
```

Here, the function no longer depends on just a single heap location, but may access an unbounded number of them. Such an access predicate under a quantifier is referred to as a quantified permission. Having the function depend on an unbounded number of heap locations means we can't apply it to a fixed number of snapshots like we did above in the case of non-quantified permissions. One way to solve this problem works as follows:

Instead of wrapping a single symbolic value into a snapshot, when we have a precondition like the one above, we instead wrap a function f_{next} : `Ref` $\rightarrow$

Ref, which, given an argument x: Ref, returns the symbolic value of x.next. This would allow us to depend on an unbounded number of heap locations.

However, functions at the SMT level are always total, so we might actually pass more heap locations than we really need. Furthermore, this would require passing a function to another function, which is not supported at the SMT level.

To make it work, we need to encode functions as regular values. To enable us to encode functions f_{id} : Ref $\rightarrow S$, we first introduce a new parametrized sort at the SMT level, called $\texttt{PSM}_S$, the sort of partial snapshot maps. We then add two functions, $\texttt{domain}_{id}$ and $\texttt{lookup}_{id}$. $\texttt{domain}_{id}$ takes as its argument a $\texttt{PSM}_S$ p representing a partial heap, and returns a set of all references x for which p contains a symbolic value for x.id. The function $\texttt{lookup}_{id}$, given a reference x and a partial snapshot map p, looks up the symbolic value of x.id in the partial heap represented by p.

If we now want to encode a concrete function f_{id}, we can introduce a new variable sm of sort $\texttt{PSM}_S$ that represents $f_i d$. We can then constrain the domain of f_{id} by placing constraints on $\texttt{domain}_{id}(sm)$. Similarly, we can tell the solver about the of value f_{id} when applied to a reference r, by asserting facts about $\texttt{lookup}_{id}(sm, r)$.

With this, we would encode an application like fun(xs), where xs is a set of references like this:

```
(fun (box_PSM_Ref sm) xs)
```

Here sm represents the function that represents every heap location that fun depends on.

Silicon also defines when two partial snapshot maps are equal, by emitting an axiom for every field that a function might require quantified permissions to. In the case of a field next of sort Ref, the axiom would look as follows, where we replaced the body of the quantification with a comment describing its meaning, for the sake of readability.

```
(assert (forall ((vs PSM_Ref) (ws PSM_Ref)) (!
    ; (domain_next(vs) = domain_next(ws) ∧
    ;    ∀x ∈ domain_next(vs) :
    ;       lookup_next(vs, x) = lookup_next(ws, x))
    ; ⟹ ws = vs
    :pattern ((box_PSM_Ref vs)
              (box_PSM_Ref ws))
    )))
```

It says that, if two PSMs have the same domain, and return the same values of every argument in that domain, then they are equal.

4.2.2 The Necessity of the Equality Axiom

It was found that some Viper files don't require the equality axiom for `PSM`s, even though they make use of heap-dependent functions that use quantified permissions. Removing the axiom makes most of them verify more quickly, in some cases up to four times faster. We modified Z3 to record how many times each quantifier is instantiated during E-matching. This showed that including the equality axiom increased the number of instantiations of almost every other quantifier. This suggests that reducing the number of instantiations of the equality axiom might have a positive performance impact for these files. We present two different approaches that aim to limit such instantiations, the first one based on enforcing an ordering on `PSM`s and the second one based on selectively emitting the axioms if needed.

4.2.3 An Ordering on PSMs

Motivation

One possible reason for the high number of instantiations of the equality axiom is its symmetry. If we can find two `PSM`s p and q to instantiate `vs` with p and `ws` with q, we can just as well instantiate `vs` with q and `ws` with p. If we define an ordering $\succ_S$ for every sort `PSM`$_S$, we could augment the trigger for the equality axiom with the term ($\succ_S$ `vs ws`). If we make sure $\succ_S$ is antisymmetric, we could not instantiate the axiom twice for the same pair of `PSM`s.

Implementation

This approach was implemented as follows. The axioms are emitted with the triggers augmented as described above. Over the course of symbolic execution, we maintain a list of every variable of sort `PSM`$_S$ we have emitted so far. Every time a new variable p of sort `PSM`$_S$ is introduced, we assert that, for every p' we emitted so far, $p' \succ_S p$ holds. p is the added to the list of previously emitted `PSM`$_S$.

Evaluation

A plot of the performance with $\succ_S$ and without is shown in figure 4.5. As we can see, it doesn't really make a difference. We offer the following as a possible explanation: When the equality axiom is instantiated with two `PSM`s p and q, the new terms this instantiation introduces to the E-matching engine give rise to a number of instantiations of other quantifiers. If the equality axiom is then instantiated with q and p, relatively few new terms are considered for E-matching, because of the axiom's symmetry. Looking at one of the files where the runtime didn't change does show a reduction

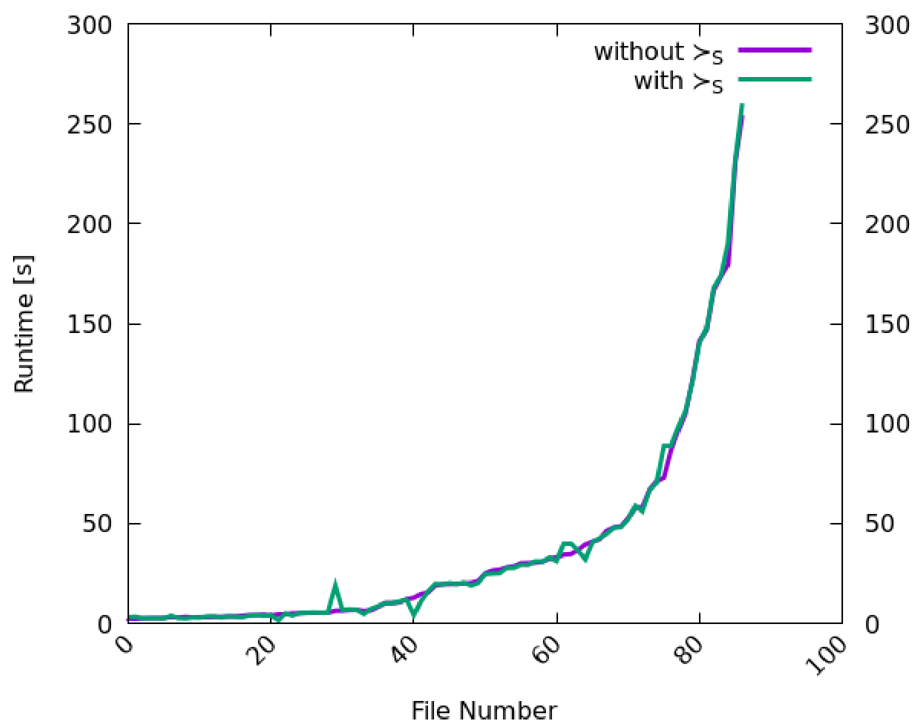

Figure 4.5: Runtime comparison on the 100 slowest files. Runtimes that exceeded the timeout of five minutes are not shown

in the number of instantiations of the equality axiom, but that didn't have any impact on the runtime.

Conclusion

All in all, this approach did not seem to have that great of an impact. It did reduce the number of instantiations, but this alone was not sufficient to lead to a performance improvement. It may be that the current implementation is inefficient. The fact that declarations in SMT-LIB persist across `(pop)` commands while assertions do not necessitates frequently reemitting a lot of assertions about $\succ_S$. It might be more efficient to do it like this: Every time we declare a new `PSM`$_S$ p, we only assert that $p' \succ_S p$, where p' is the `PSM`$_S$ that immediately precedes p. This would cut down on the number of assertions that need to be reemitted after a `(pop)`. In addition we would assert that $\succ_S$ is transitive. Asserting transitivity in an efficient manner is not entirely trivial, but possible.

4.2.4 Dynamically Tightening Triggers

Motivation

Recall how an application of a heap-dependent function is encoded at the SMT level. Where `sm` is a partial snapshot map and `xs` a set of references, the application `fun(xs)` looks like this:

```
(fun (box_PSM_Ref sm) xs)
```

With this in mind we can try to come up with tighter triggers for the equality axioms by including the function application in the trigger. This means that we now have to emit seperate versions of the axioms for every function, but each is more specific. For the function `fun`, we would emit the following axiom:

```
(assert (forall ((vs PSM_Ref) (ws PSM_Ref) (xs1 Set_Ref) (xs2 Set_Ref)) (!
    ; the body stays the same
    :pattern ((fun (box_PSM_Ref vs) xs1)
              (fun (box_PSM_Ref ws) xs2))
    )))
```

However, tightening the triggers like this makes some tests fail to verify. One such failing example is this:

```
field f: Int

function foo(xs: Set[Ref], i: Int): Bool
  requires forall x: Ref :: {x in xs} x in xs ==> acc(x.f)

method test(xs: Set[Ref]) {
  inhale forall x: Ref :: acc(x.f)
  assume forall i: Int :: foo(xs, i)
  assert forall i: Int :: foo(xs, i)
}
```

The assertion on the last line obviously holds, but Silicon can't prove it if we use the tigher triggers. We studied the SMT-logs produced during verification, and couldn't find a reason why the solver can't establish that this holds.

With this in mind, we see that some files don't require the equality axioms at all, and verify more quickly without them, some others need them but can tolerate tighter triggers, and some require the liberal default triggers. We can't statically tell which category a particular file falls into. We propose a mechanism that dynamically retries failed assertions with increasingly liberal triggers, with the hope that we might often get away without the axioms, and can still have them when we need them.

Mechanism

Figure 4.6 provides an overview of the retry mechanism. Normally, the axioms are not emitted at all. If the solver fails to prove a particular assertion, we immediately emit the axioms with tight triggers and try again. If that's still not enough, we emit the most liberal version of the Axiom. If this still doesn't work, we accept that the solver can't show what we wanted to know. If that happens, Silicon might trigger a state consolidation and try the assertion again. In that event, we immediately emit the most liberal axioms.

Generating Tighter Triggers

Coming up with the tight triggers is not always as easy as in the case of `fun` above. If the funcion precondition contains accesses to multiple fields or conditional expressions, the `Snap` that is passed on application becomes more complicated. Fortunately, the structure of the `Snap` can be determined from the functions preconditions. If the precondition contains conditional expressions, the function may be passed differently structured `Snap`s depending on the condition. Consider the following function:

```
field f: Int
function fun02(xs: Seq[Ref]): Int
  requires |xs| > 10
  requires forall k: Int :: 0 <= k && k < 3 ==> acc(xs[k].f)
  requires forall k: Int :: 6 <= k && k < 9 ==> acc(xs[k].f)
```

An application `fun02(xs)` looks like this at the SMT-Level:

```
(fun02 (pair
  unit
  (pair
    (box_PSM_Int sm_1)
    (box_PSM_Int sm_2))) xs)
```

If we want to emit the axiom with tight triggers for this function, we need to emit it twice, once each of the partial snapshot maps in the `Snap` tree. If we emit the axiom for one of the partial snapshot maps, we need to additionally quantify over the other. For `fun02`, one of the axioms would look like this:

```
(assert (forall
  ((vs PSM_Ref) (ws PSM_Ref) (xs_1 Seq_Ref) (xs_2 Seq_Ref) (s_1 Snap) (s_2 Snap)) (!
    ; the body stays the same
    :pattern
      ((fun02 (pair unit (pair (box_PSM_Int vs) s_1)) xs_1)
       (fun02 (pair unit (pair (box_PSM_Int ws) s_2)) xs_2))
    )))
```

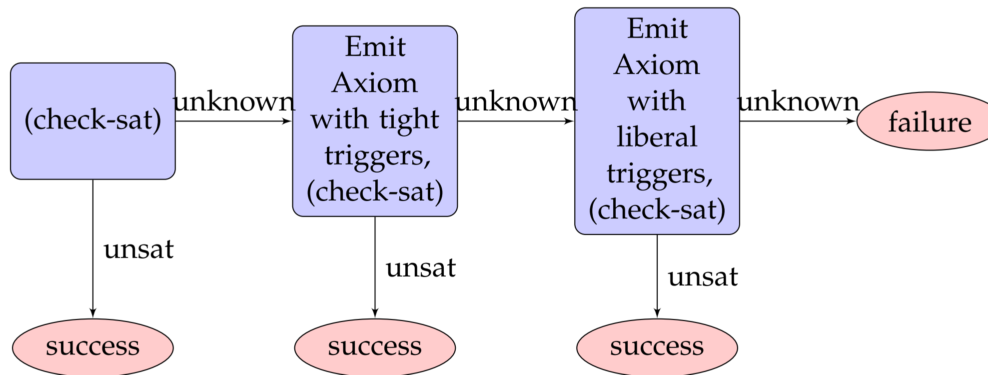

Figure 4.6: Overview of the Retry Mechanism

Evaluation

Figure 4.7 shows a comparison of runtimes for runs with retrying and runs without, where just the default axioms were emitted. The two downward spikes, where retrying paid off significantly, are two constructed examples that are slowed down by the presence of the axioms. When verifying these two examples, no retries were necessary at all. We also tried the approach on a few longer examples from a project about flow verification [6], the timings are given in figure 4.8. In this case the retrying strategy seems to pay off. However, Carbon verifies each of these files in under a minute.

Conclusion

The approach seems to work under some circumstances. However, it doesn't seem to be beneficial in general. Also, when it does work, it only goes so far. As we can see with the flow verification examples, there must be more at play. It also introduces coupling between different parts of the code. If the structure of `Snap` trees changes, the method that computes the tighter triggers would need to be adapted as well. The coupling of the structure of function predicates and `Snap` trees in general is not desirable. The whole approach to heap-dependent function axiomatization is currently being worked on, and it is not clear yet if this approach will still be relevant once that is completed.

4.3 Weak Memory Verification

In this section we'll have a look at the performance problems exhibited by a couple of examples from a project about weak memory verification [12].

We have a total of 14 files, of which 10 exhibit major performance problems. A cursory investigation reveals that, for all 10 files, Z3 gets stuck on a `(push)` command.

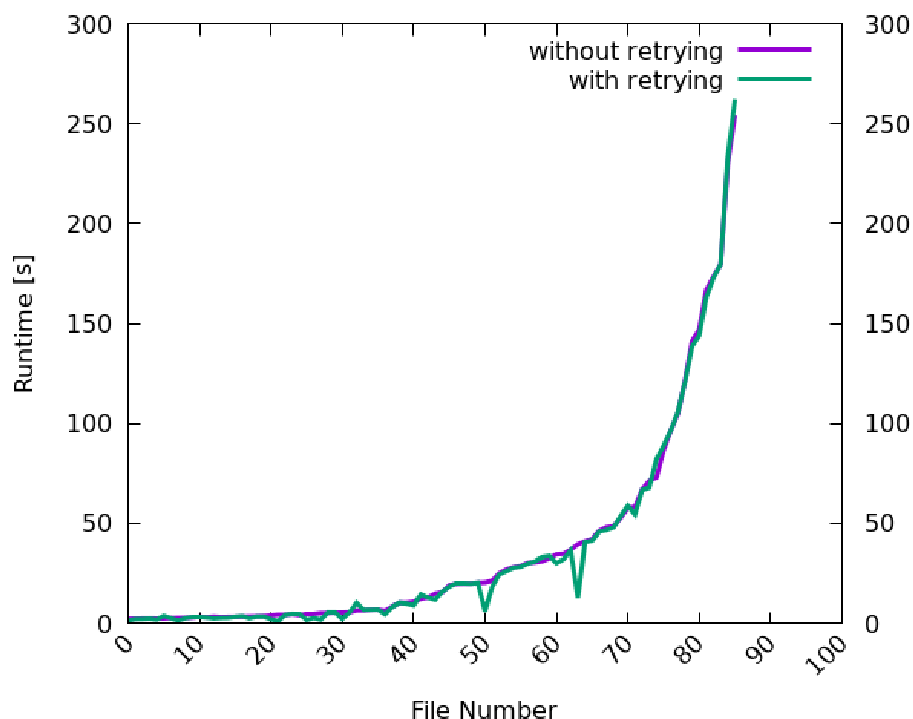

Figure 4.7: Runtime comparison of the 100 slowest files. Runtimes that exceeded the timeout of five minutes are not shown

Flow Verification

File	Before	After
pip.vpr	>25m	14m
composite.vpr	19s	13s
list.vpr	12m	5m
dll.vpr	42m	9m

Figure 4.8: Runtime comparison on flow verification files

At first this seems counterintuitive. After all, pushing a new scope on the assertion stack shouldn't really cost anything. However, it makes sense to put in some work to learn all we can from the current stack before pushing a new scope. That way, when we pop the scope later, we can throw away everything we learned between the push-pop pair. If we didn't make the effort to get everything out of the current stack before pushing, we might later learn new facts that don't depend on the topmost scope. We wouldn't want to throw those away when we pop the scope, because they are still valid without the topmost scope. We could try and keep track of what scopes everything depends on, but it is simpler to just do some work before pushing a new scope. While this is an implementation detail of Z3, and the problem

would probably just manifest at the next `(check-sat)` if Z3 handled pushing differently, we still feel it is useful to be aware that satisfiability checks aren't the only thing where Z3 might take a long time.

4.3.1 What's Happening

If we look at what Z3 is doing, for example by attaching a debugger and interrupting it periodically, we see that it spends a lot of time in the E-matching engine. This, coupled with the rapidly increasing memory consumption, suggest that we may have a matching loop on our hands.

We are going to use the Axiom-Profiler to diagnose these loops. The Axiom Profiler is a "tool that enables users to analyse instantiation problems effectively, by filtering and visualising rich logging information from SMT runs."[2] It can also detect potential matching loops and offer explanations as to why they loop. To diagnose a matching loop, we first obtain an SMT log file from a run of Silicon. We then remove everything that's in a scope that will be popped off the stack by the point where the problem occurs. We also remove all satisfiability checks in the file. These two measures are intended to isolate whatever it is that causes problems for Z3. It also reduces the size of the resulting Z3 trace. Then we rename every quantifier that still remains in the file, such that each quantifier has a unique identifier. This is done so we can easily find the offending quantifiers in the SMT-LIB code if the Axiom Profiler implicates them in a matching loop. Then we run Z3 on the modified SMT-LIB script using the command line arguments trace=true and proof=true, as per the instructions from the Axiom Profiler repository [13].

Z3 will enter the matching loop, and should be killed quickly, otherwise the resulting trace might be too large for the Axiom Profiler to handle efficiently. The Axiom Profiler can then be started on the trace file. In the Axiom Profiler a bit of manual searching is required to find a quantifier instantiation chain that's involved in a matching loop and get the Axiom Profiler to run its matching loop detection on it.

4.3.2 The Matching Loops

We found matching loops in every one of the 10 problematic files. The loops can be classified into three classes, depending on the quantifiers involved. Figure 4.9 shows a representative member of each of the three classes, visualized using the Axiom Profiler's quantifier blame visualization. A complete listing of the looping quantifiers for each file can be found in the appendix in figure .1.

These quantifiers are generated by Silicon. The quantifiers named `prog.lx-aux` take their triggers from the Viper program to verify, while the other two get

Figure 4.9: Representatives for each of the three matching loop classes

their triggers from Silicon. We weren't able to come up with any triggers for the Viper programs that avoid these matching loops. Looking at the line numbers embedded in the `prog.l`x`-aux` quantifiers, we can see that the self loops seem to have something to do with the `FENCEAcq` macro. The loops between the `prog.l`x`-aux` and `qp.fvfValDef`x quantifiers can similarly be traced to the `fetchUpdate` macro.

4.3.3 Trying to avoid the Matching Loops

One thing we can notice is that the problematic `prog.l`x`-aux` are always triggered by a single function application (`temp<Ref> r`), where r is the quantified variable. We managed to avoid getting stuck in the scope pushes by splitting up quantifiers and tightening some of the (`temp<Ref> r`) triggers. For the quantifier splitting, we transformed any quantifier of the form

```
(forall (qs) (! (and e0 ... en)))
```

into

```
(and (forall (qs) (! e0)) ... (forall (qs) (! e1)))
```

Additionally, we flattened any nested applications of `and`, and applied the transformation from the inside out, so we could completely split nested quantifiers. This splitting enables selectively tightening the triggers for some of the quantifiers. Then we take any quantifier that is triggered by (`temp<Ref> r`) and check if its body contains an lookup of a predicate in a predicate snapshot map. What this means is not really important here, we just looked for a larger term involving (`temp<Ref> r`) to use as a tighter trigger. We put this transformation in a script that can sit between Silicon and Z3 and perform the transformation on the fly. Since this is just a small experiment that's very specific to the cases at hand, there's no need to do it in Silicon.

We find that applying this transformatin gets all 10 cases past the problem-

atic scope pushes, and one of them even manages to verify successfully, although it does take about 15 minutes to do so. The other files appear to get stuck at some later satisfiability check. It could not be determined if this was due to matching loops as well, since the Axiom Profiler could no longer handle the traces produced by Z3, but it seems likely based on inspection under a debugger. This is not surprising as the transformation is very specific to the matching loops that were found initially.

4.3.4 Conclusion

We showed that the performance problems of a group of examples was caused by matching loops. We hope our describing the process we used to find them can be helpful for others that find themselves confronted with matching loops. Additionally, we showed that these particular loops can be avoided, but more work is required in this area. Since some of the triggers involved in the loops came from the Viper programs, it may be beneficial to try and recognize if a user-provided trigger might interfere with Silicon's internal triggers. There has been work on detecting and preventing matching loops in the Dafny program verifier [7]. The ideas from this paper might be applicable to Silicon as well.

Chapter 5

Solver Interface

One potential source of inefficiency is the way the symbolic execution engine interacts with the SMT solver. This interaction can happen in one of two ways.

On one hand, the symbolic exection engine can generate formulas and commands in the SMTLib language that the SMT solver can interpret. This text is sent to the standard input of the solver, and its standard output is read to receive the result. This method is referred to as the StdIO approach. Potential downsides of this approach include the overhead of first serializing structured data to text on the symbolic exection side, and then parsing that text on the solver side.

The other approach is to communicate with the solver via an application programming interface (API). In this approach, the solver provides bindings to a variety of its functions. The symbolic execution engine may then control the solver by calling these functions. Furthermore, the symbolic execution can internally use the solver's datastructures for formulas, thus avoiding any transliteration of data.

Changing Silicon to fully utilize Z3s Java bindings would require a lot of changes to Silicon if we want to get rid of any needless translations from Silicons internal datastructures to Z3s internal datastructures. Thus we will just try and find out if the changes will be worth it without implementing them. To do this, we make use of another symbolic execution based verifier, VeriFast. VeriFast already offers us the choice whether to use StdIO or an API to interact with the solver. We can run VeriFast on its own testsuite and compare the runtime of text- vs API-based interaction.

A few caveats apply here. First, Z3 doesn't seem to be the preferred solver to use with VeriFast, as most of the test cases use the Redux solver by default. Given that VeriFast supports different solver APIs, it is safe to assume that it still includes a translation step from its internal format to one that is

compatible with Z3s API. Furthermore, most of VeriFasts test cases are very short, which means that the difference in performance between StdIO and API may be lost in random performance fluctuations. Meanwhile, other test cases take a long time to verify when using Z3, and some exhibit runtime differences that are way beyond what could be explained by the differences caused by StdIO and API.

5.1 Measurement

Measurement was carried out in the following steps

- VeriFast includes a program to run the testsuite, called mysh. mysh was modified so as to print the time taken by failed testcases, and continue with the testsuite even if some tests fail. This was necessary because some of the tests that pass using the Redux solver fail when using Z3.
- The testsuite itself was modified to exclude all testcases that take an unreasonably long time to execute when using Z3.
- The testsuite recursively executes smaller testsuites. A script to flatten this recursion into one larger testsuite was written. The script also modifies all testcases to use either the Z3 API or Z3 over StdIO.
- With that we can run the modified mysh on the modified testsuite and capture its output for further processing.

5.2 Results

The results for all tested files are shown in figure 5.1. The runtime for StdIO (t_{IO}) is shown in blue and the runtime for the API (t_{API} is shown in green, both against the axis on the right. The ratio $\frac{t_{IO}}{t_{API}}$ is shown in red against the left axis. While the ratio suggests that the API was faster, most of the test cases were very short.

In figure 5.2 we plot the top 30 testcases in terms of how long they took when using StdIO. As the testcases take longer, the speedup seems to settle around 2x. One interesting outlier is predctors.c, where the API takes a lot longer. A reason for that might be that the verification proceeded differently depending on whether StdIO or the API was used. This is supported by the fact that, when the SMTLib code produced for StdIO was recorded and compared to an SMTLib log file produced with the API, the code was different.

All in all, the data suggests that using an API to interact with the solver could certainly be worth the effort of implementing it.

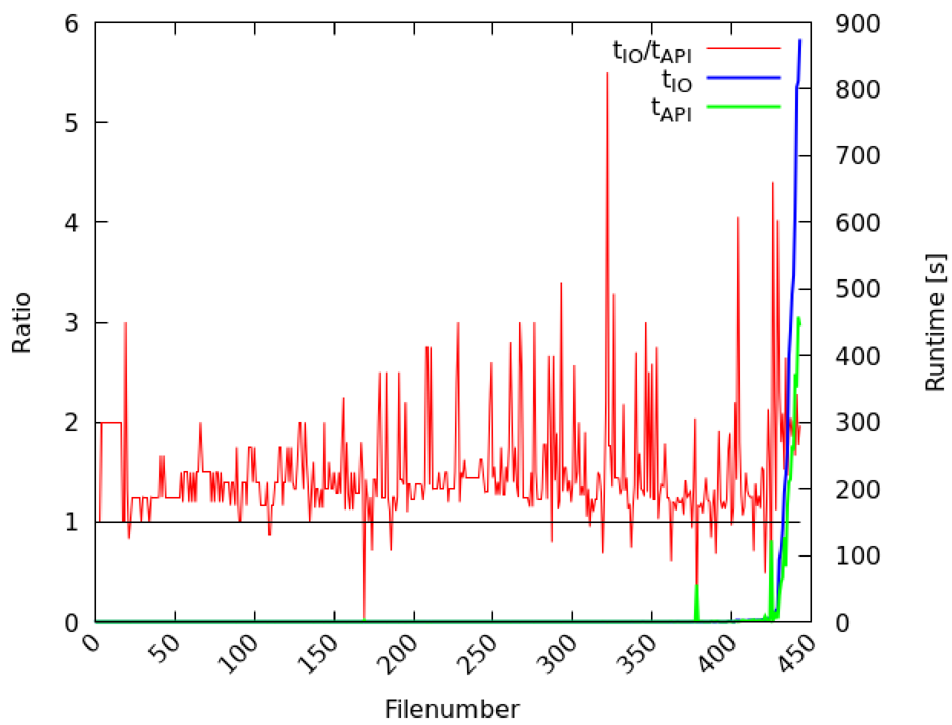

Figure 5.1: Comparison of VeriFast runtimes using StdIO or API

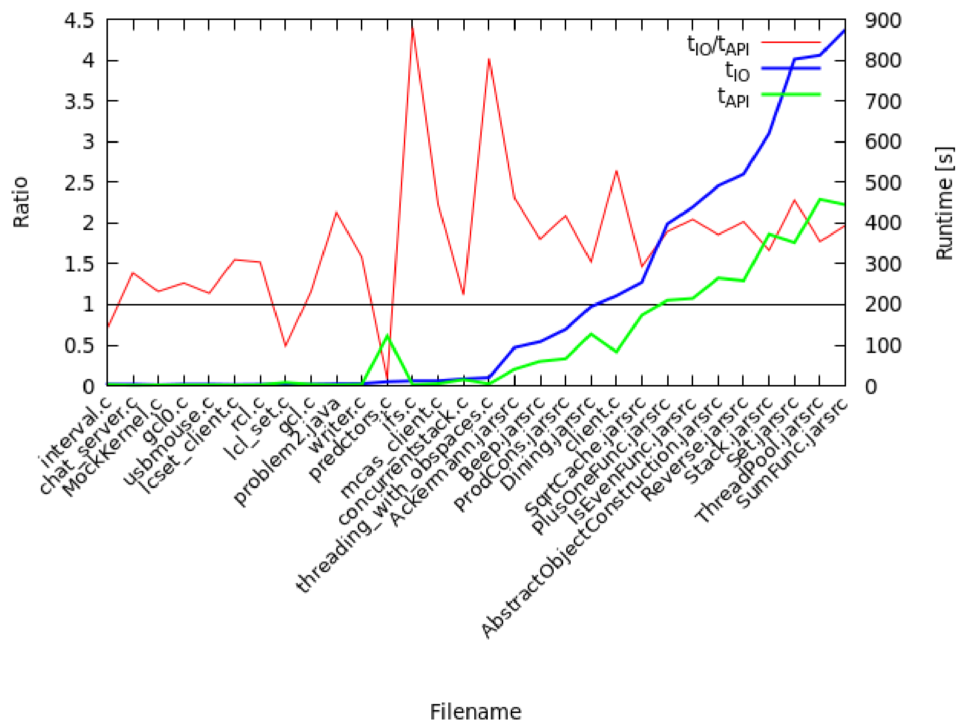

Figure 5.2: Comparison of VeriFast runtimes using StdIO or API, slower cases only

Chapter 6

Resource Bounds

6.1 Background

During symbolic execution, it's often the case that knowing if a particular proposition about the current state holds would be beneficial for performance reasons, but not crucial to the verification outcome. We might for instance want to check if a branch condition is known to be always true, which would allow us to skip exploring the path associated with the condition being false. However, if the SMT-solver takes too long to establish that the condition is always true, we might spend more time waiting for an answer than it would have taken us to just explore the path in question. In such circumstances, Silicon uses timeouts to bound the solver's execution time. Z3 also offers the option to set a resource limit instead of a timeout. Z3 assigns resource costs to various basic operations, and will stop if the accumulated cost exceeds the limit we set. The amount of resources used between different runs of Z3 on the same SMT-LIB input is always the same, even on different machines.

6.2 Problems with Timeouts

Verification Instabilities

Using timeouts may cause certain queries to return different results in repeated runs, if the time it takes to solve the query is about equal to the timeout. This won't affect the verification outcome, but can affect the runtime, if say a path is sometimes explored because the solver ran into a timeout, and sometimes skipped because the solver was just fast enough.

Comparing Different Machines

Timeouts make it trickier to compare the performance of Silicon accross different computers. Checks that always work on one machine may never succeed on another, slower computer. For instance, on an Intel i5-4210M, at 2.6 GHz, Z3's resource counter increases by between 20000 and 30000 units in 10 milliseconds, whereas it only increases by between 15000 and 19000 in the same time on an Intel Core2 Duo at 2.53 GHz.

6.3 Implementing Resource Limits

Replacing timeouts with resource limits throughout the codebase should not take too much work. The main difficulty will be coming up with the right magic numbers to replace the existing default timeouts. It seems questionable to base the resource limits too much on the existing timeouts, given how the amount of resources used in a given time is rather variable across different machines.

Chapter 7

Miscellaneous

7.1 Z3's Parallel Mode

As of version 4.8.0, Z3 features a parallel mode for select theories [15].We tried enabling it during the initial verification of functions and predicates, which always uses only one Z3 instance, as well as enabling it as soon as all but one verification task had finished. However, no difference in performance was found, and looking at Z3's CPU utilization revealed it never used more than one core when in parallel mode. This is most likely due to the theories that Silicon requires not being supported by the parallel mode.

7.2 Bug Fixes

Over the course of this work, a few bugs were found both in Silicon and other projects.

7.2.1 Silicon

A bug was fixed where some operations on sets and multisets on the viper level were typed incorrectly, which led to Silicon rejecting valid programs if they used these operations. As an example, the result type of the subset predicate $A \subset B$ was declared to be another Set when it should have been a Boolean.

7.2.2 VeriFast

When attempting to compare the performance of using an API or StdIO for solver interaction (see Chapter 5), it was found that VeriFast's SMTLib output did not conform to the SMTLib 2.0 Standard. VeriFast would render Fractions as `1/2` instead of `(/ 1 2)`, or emit identifiers that were reserved by Z3, such as `store`, or not valid according to the standard, such

as `64bitcount`. This was fixed in order to proceed with the measurements, and the changes have since been accepted into VeriFast.

7.2.3 AxiomProfiler

Some bugs in the AxiomProfiler were found and fixed locally. They have been brought to the attention of the maintainers.

Chapter 8

Future Work

This thesis presents a lot of opportunity for follow-up work. We make a strong case for using Z3's API instead of using StdIO to communicate. We imagine that implementing this in the most optimal way would be a rather large change to Silicon. It could also be done in a more encapsulated manner by translating Silicon's internal representation to the representation used by the API.

Further investigation of the collected examples could be done. This thesis mainly focused on some and paid comparatively little attention to others. We want to specifically mention the files provided by Marco Eilers (In the folder `me/` in [3]), as profiling them indicated that they spent comparatively less time in Z3 than the other examples.

Something else that could be addressed is the issue of matching loops. It would be nice if Viper could support the user in coming up with appropriate triggers for their quantifiers.

Appendix

Filename	Looping Quantifiers
FencesDblMsgPassAcqRewrite.sil	prog.l125-aux with itself
	prog.l78-aux with itself
FencesDblMsgPass.sil	prog.l120-aux with itself
	prog.ll71-aux with itself
FencesDblMsgPassSplit.sil	prog.l121-aux-309 with itself
	prog.l74-aux-309 with itself
FollyRWSpinlock_err_mod.sil	No matching loop
FollyRWSpinlock_err.sil	prog.l81-aux qp.fvfValDef51
	$Snap.$Ref qp.fvfValDef57
	prog.l128-aux-390 qp.fvfValDef288
	$Snap.$Ref qp.fvfValDef221
FollyRWSpinlockStronger_mod.sil	No matching loop
FollyRWSpinlockStronger.sil	$Snap.$Ref qp.fvfValDef78
	prog.l68-aux qp.fvfValDef56
	$Snap.$Ref qp.fvfValDef286
	prog.l122-aux qp.fvfValDef222
RelAcqDblMsgPassSplit.sil	No matching loop
RelAcqMsgPass.sil	No matching loop
RelAcqRustARCStronger.sil	prog.l73-aux qp.fvfValDef85
RSLLockNoSpin-not-in-appendix.sil	qp.fvfValDef115 $Snap.$Ref
	prog.l30-aux qp.fvfValDef23
RSLSpinlock.sil	$Snap.$Ref qp.fvfValDef87
RustARCOriginal_err.sil	prog.l75-aux qp.fvfValDef85
	prog.l57-aux qp.fvfValDef44
RustARCStronger.sil	prog.l74-aux qp.fvfValDef85
	prog.l57-aux qp.fvfValDef44

Figure .1: The Matching Loops in the Weak Memory Verification Files

Bibliography

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[2] Nils Becker, Peter Müller, and Alexander J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116, Cham, 2019. Springer International Publishing.

[3] Mauro Bringolf, Linard Arquint, and Moritz Knüsel. Slow Examples. https://bitbucket.org/maurobringolf/silicon-examples/src/master/. Accessed: 2019-09-13.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] Brendan Gregg. bpftrace (DTrace 2.0) for Linux 2018. http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html. Accessed: 2019-09-13.

[6] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. Slow Examples. Draft obtained from http://people.inf.ethz.ch/summersa/.

[7] K. R. M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 361–381, Cham, 2016. Springer International Publishing.

[8] LWN.net. A thorough introduction to eBPF. https://lwn.net/Articles/740157/. Accessed: 2019-09-13.

[9] Jakub Łopuszański Michał Moskal and Joseph R.Kiniry. E-matching for fun and profit. *Electronic Notes in Theoretical Computer Science*, 198(2):19 – 35, 2008. Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007).

[10] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[11] Malte Schwerhoff. Advancing automated, permission-based program verification using symbolic execution, 2016.

[12] Alexander Summers and Peter Müller. *Automating Deductive Verification for Weak-Memory Programs*, pages 190–209. 04 2018.

[13] Viperproject. Axiom Profiler Repository. https://bitbucket.org/viperproject/axiom-profiler/src/default/. Accessed: 2019-09-13.

[14] Viperproject. Viper Tutorial. http://viper.ethz.ch/tutorial/. Accessed: 2019-09-13.

[15] Z3-Prover. Z3 Release Notes. https://github.com/Z3Prover/z3/blob/master/RELEASE_NOTES. Accessed: 2019-09-03.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Optimization of a Symbolic-Execution-Based Verifier

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Knüsel	Moritz

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Abtwil, 12.5.17	Moritz Knüsel

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.