Bachelor's Thesis

Performance Improvements of a Program Verifier

Fabian Bösiger
Supervised by Dr. Malte Schwerhoff

Programming Methodology Group
Department of Computer Science
ETH Zürich

August 3, 2021

Abstract

This thesis explores ways to improve performance of the Silicon program verification backend for the Viper verification infrastructure.

In the first part, we explore the concept of applying the flyweight pattern on ASTs. Applying the flyweight pattern avoids multiple instances of structurally equal nodes existing at the same time. This allows us to replace structural and recursive equality checks with reference equality checks, with the goal of improving performance of equality checks.

In the second part, we introduce more sophisticated ways to join symbolic execution paths in Silicon, after having branched on conditional expressions, implications and if-statements.

Acknowledgements

I would like to thank my supervisor, Malte Schwerhoff, who provided me with the opportunity to write this thesis. The time and effort he expended to help and advice me was highly appreciated.

Additionally, I'd like to thank Peter Müller and his group for sparking interest and providing insight into topics such as these.

Furthermore, I'd like to express my gratitude towards my family for providing me with a very pleasant home office environment.

Contents

1 Introduction

Viper [5] is a verification infrastructure on top of which verification tools for different programming languages can be built. Silicon [7] is a backend for Viper, which is based on Smallfoot-style [1] symbolic execution. To advance program verification in practice, fast verification is crucial as it provides a more streamlined experience for developers. This is the reason why one of Silicon's stated goals is performance:

> "The verifier should enable an IDE-like experience: it should be sufficiently fast such that users can continuously work on verifying programs [...]" [7]

In this thesis, we explore two different approaches to improving performance of Silicon.

First Approach

Silicon internally uses abstract syntax trees (AST) to represent the structure of a program as a tree data structure. As with any other tree structure, ASTs can be traversed, searched, transformed and so forth. During such operations, subtrees within the AST are potentially checked for equality many times. Moreover, equality checks also occur in operations on collections of AST subtrees, for example in finding a specific subtree, which may add additional performance overhead.

Equality checks can't easily be avoided, but they can be implemented in a more performant way. Silicon's AST nodes are called terms, which represent different program operations. `Plus(IntLiteral(1), IntLiteral(2))` represents the program code $1 + 2$. `Plus`, `IntLiteral(1)` and `IntLiteral(2)` are all terms. Currently in Silicon, new term instances are created independently of already existing ones, which potentially leads to the coexistence of multiple structurally equal term instances. Subterm equality is checked in a structural and recursive manner. In part I of working towards a potential improvement in performance, we explore the concept of applying the flyweight pattern [2] on AST terms to only ever have one instance of some

term structure, thus avoiding the need for structural and recursive equality checks.

Second Approach

For verifying a program, Silicon uses the symbolic execution approach, where the program is interpreted, and a symbolic state keeps track of all possible program states at the current point of execution, for all possible input values of the program. When encountering certain expressions or statements, for example an if-statement, symbolic execution branches with the assumptions of the corresponding program path.

Silicon currently only joins these branches for some simple cases. In other cases, branches aren't joined, which results in all statements later down the verification path being evaluated essentially twice, but with different assumptions in each branch. Both of these verification paths may branch again, potentially leading to exponential growth in the number of branches. In an effort to improve performance, part II of this thesis focuses on implementing joining of execution paths or more complex cases, which ultimately leads to fewer active branches.

Part I

Flyweight ASTs; A Study in Applied Laziness

2 Approach

2.1 Implementation of Flyweight ASTs

Currently in Silicon, new term instances are created independently of already existing ones, which potentially leads to the coexistence of multiple structurally equal term instances. Subterm equality is checked in a structural and recursive manner. However the AST used in Silicon is immutable, so the flyweight pattern [2] can be applied on AST terms. To do this, a pool of term instances is maintained. Whenever a term is to be created, the components of this new term is compared with the pool of existing terms. If a term with the same components already exists, a reference to the existing term is returned and the creation of a new instance is avoided. Otherwise, a new term is created and added to the pool.

This gives the guarantee that there are no two instances of the same term in our pool, meaning every two structurally equal terms point to the same underlying object in memory. Comparing terms for structural equality then boils down to a cheap reference equality check, and recursive equality checks can be avoided, at the cost of increased overhead at the creation of a term instance due to the flyweight pattern.

2.2 Automate Boilerplate Generation Using Macros

Silicon's AST representation of the Viper language consists of nearly 100 different terms, all with boilerplate implementations for different operations. For example, because Silicon usually doesn't use case classes for its terms, each term defines it's own `unapply` method. Our changes adds additional boilerplate code by implementing the flyweight pattern for each term as seen in listing 3.1.

Our ASTs shouldn't only be flyweight in the sense of the implementation pattern, but also regarding development time and effort. The Viper infrastructure is written in the Scala programming language, which provides seamless interoperability with Java and has support for metaprogramming using macros. [6] This allows us to avoid boilerplate code and instead automatically generate it using Scala's support for macro annotations. Additional benefits of using macro annotations include improvements in code readabil-

ity and maintainability. Experimenting with code changes will become a matter of editing a single macro instead of editing each term individually. Terms which may be added in the future are also easier to implement.

3 Implementation

3.1 Implementation of the Flyweight Pattern

As the flyweight pattern for Silicon's terms is implemented in Scala, we assume general familiarity with the language. The implementation works as follows:

1. The constructor of a term is made private (line 1) so that new term instances can't be created via the `new` keyword, but only via the `apply` method (line 9), which acts as the term factory and does the pool lookups.
2. For every term, we create a map which maps the components of the term to the term itself (line 7). This allows us to later look up whether a structurally equal term was created already (line 10).
3. In the `apply` method, we check the pool for structurally equal instances (line 10), and if one exists, we return it and thus avoid creating a new instance of the same term (line 20).
4. If no structurally equal instance exists, we create a new instance via the `new` keyword, add it to the pool and return it (line 14, 15, 16).

As an example, the implementation of the flyweight pattern for the `Plus` term is shown here:

Listing 1: Implementation of the flyweight pattern.

```
1  class Plus private (val p0: Term, val p1: Term) {
2      // ...
3  }
4
5  object Plus extends ((Term, Term) => Term) {
6      // Maps fields of the term to the term instance itself.
7      var pool = new Map[(Term, Term), Term]
8
9      def apply(e0: Term, e1: Term): Term = {
10         pool.get((e0, e1)) match {
11             // If no structurally equal term exists,
```

```
12              // create a new one.
13              case None =>
14                  val term = new Plus(e0, e1)
15                  pool.addOne((e0, e1), term)
16                  term
17              // If a structurally equal term exists,
18              // return a reference to it instead.
19              case Some(term) =>
20                  term
21          }
22      }
23
24      // ...
25  }
```

3.2 A Macro Annotation for Code Generation

The Viper infrastructure is written in the Scala programming language, which has support for metaprogramming using macros. Silicon's different AST term classes are an ideal target for static code generation, as they inherently share many similarities with each other, their code is structurally equivalent, but differ in type and arity. To address the problem of boilerplate code described in section 2.2, we implement a macro annotation that automatically generates required code.

The code for the flyweight macro annotation exists as a subproject within Silicon. Each term can be annotated with `@flyweight`, which invokes the macro at compile time and rewrites the term in the following way:

1. If an `apply` method is already defined, rename it to `_apply`. The already defined `apply` method can't be discarded because it potentially defines additional operations required on creation of a term.

2. Define a new `apply` method that introduces the flyweight pattern as discussed in section 3.1. If a new term instance has to be created, either use the previously defined `_apply` method if it exists, else simply create an instance using the `new` keyword.

3. Generate a suitable `unapply` method.

4. Generate a `copy` method that calls `apply` instead of creating instances via `new` such that the flyweight pattern can't be bypassed when copying a term.

5. Override `hashCode` to use Java's `System.identityHashCode`.

This process of rewriting terms using macros happens on every term annotated with `@flyweight`, and can in nicely illustrated by an example that considers the program input in listing 2, and output in listing 3 of our macro:

Listing 2: Input code annotated with the macro.

```
1  @flyweight
2  class Plus(val p0: Term, val p1: Term)
3      extends ArithmeticTerm
4  {
5      override val op = "+"
6  }
7
8  object Plus extends ((Term, Term) => Term) {
9      def apply(e0: Term, e1: Term): Term = (e0, e1) match {
10         case (t0, Zero) => t0
11         case (Zero, t1) => t1
12         case (IntLiteral(n0), IntLiteral(n1)) => IntLiteral(n0 + n1)
13         case _ => new Plus(e0, e1)
14     }
15 }
```

Listing 3: Output code generated by our macro.

```
1  class Plus private (val p0: Term, val p1: Term)
2      // Superclasses and implemented traits are preserved from input.
3      extends ArithmeticTerm
4  {
5      // Override hashCode.
6      override val hashCode = System.identityHashCode(this)
7
8      // Generated copy method which uses the generated apply method.
9      def copy(p0: Term = p0, p1: Term = p1) = Plus(p0, p1)
```

```
10
11      // Preserved from input.
12      override val op = "+"
13  }
14
15  object Plus extends ((Term, Term) => Term) {
16      var pool = new Map[(Term, Term), Term]
17
18      // Define new apply method which uses the flyweight pattern.
19      def apply(e0: Term, e1: Term): Term = {
20          pool.get((e0, e1)) match {
21              case None =>
22                  val term = Plus._apply(e0, e1)
23                  pool.addOne((e0, e1), term)
24                  term
25              case Some(term) =>
26                  term
27          }
28      }
29
30      // Generated unapply method.
31      def unapply(t: Plus) =
32          Some((t.p0, t.p1))
33
34      // Renamed existing apply method to _apply.
35      // AST simplifications implemented are thus preserved.
36      def _apply(e0: Term, e1: Term): Term = (e0, e1) match {
37          case (t0, IntLiteral(0)) => t0
38          case (IntLiteral(0), t1) => t1
39          case (IntLiteral(n0), IntLiteral(n1)) => IntLiteral(n0 + n1)
40          case _ => new Plus(e0, e1)
41      }
42  }
```

3.3 Flyweight Macro Support for IntelliJ

The Viper infrastructure is written in the Scala programming language. Scala has support for metaprogramming using macros, which provide a nice

and easy way for metaprogramming and are regularly used. For a nice programming experience using macros, IDE support should ideally be provided. In this case, we use the IntelliJ IDE. However, coding assistance for Scala macros is not supported natively by the IntelliJ IDE, as it is difficult for IDE's to provide proper syntax highlighting.:

> "Since IntelliJ IDEA's coding assistance is based on static code analysis, the IDE is not aware of AST changes, and can't provide appropriate code completion and inspections for the generated code." [3]

In the example of our flyweight macro, the IDE is not aware that the method `apply` is generated and exists in the by our macro expanded code. The IDE thus reports an error that the method `apply` doesn't exist wherever a term is applied, despite `apply` existing in the expanded code, as illustrated in figure 1.

To fix this issue for the IntelliJ IDE, we provide a plugin which can be easily installed in IntelliJ, and fixes the highlighting issues for the flyweight macro. The plugin is hard-coded to make the IDE aware of code changes introduced by the flyweight macro, as seen in figure 2.

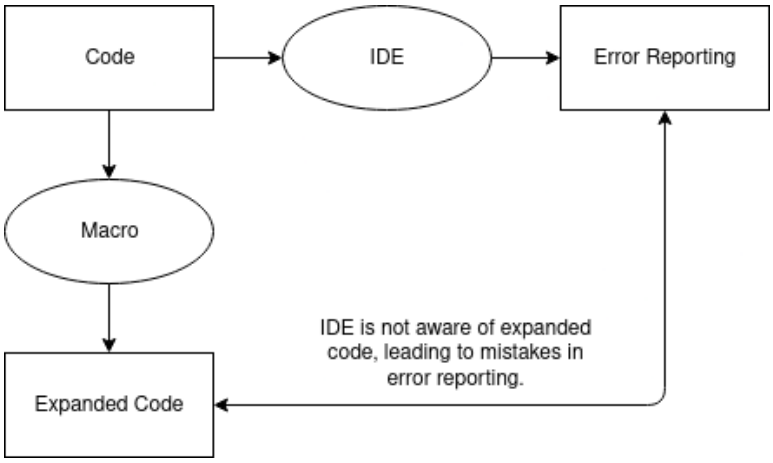


Figure 1: The IDE is not aware of code changes done by our macro. This leads to incorrect error reporting.

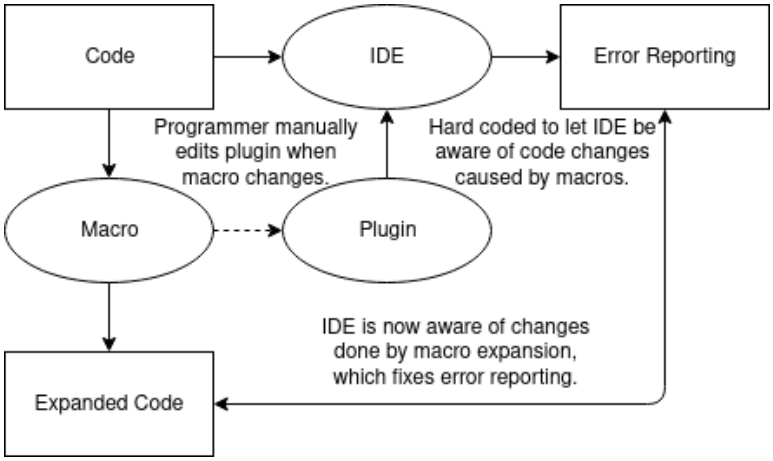


Figure 2: The IDE is now aware of code changes done by the flyweight macro, which fixes error reporting.

3.3.1 Towards Full Scala Macro Support for IntelliJ

The plugin discussed in section 3.3 is hard-coded to only support the flyweight macro. Changes in the flyweight macro require the programmer to manually modify the plugin. To encourage more experimentation using macros, it is of advantage to have a plugin which supports macros that may be modified, for example by generating additional methods.

To support this kind of more dynamic plugin, the macro is modified to dump all generated method signatures into a configuration file. This configuration file is then read by the IntelliJ plugin, which now knows the signatures of the methods generated by the macro. Whenever the macro is modified and the program is compiled again, the configuration file is rewritten, and the IntelliJ plugin is aware of the changes. This is illustrated in in figure 3.

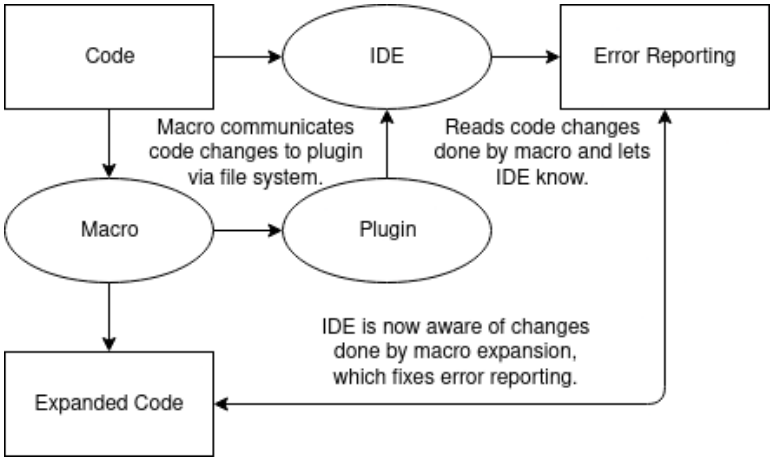


Figure 3: The IDE is now aware of code changes done by the flyweight macro, which fixes error reporting.

4 Evaluation

In the following sections, we discuss the performance impact of introducing the flyweight pattern to Silicon's AST. The Silicon implementation without flyweight ASTs will be referred to as the base implementation.

In section 4.1, we evaluate the performance difference of using different map implementations for the pool holding all term instances, seen in listing 1, line 7. We further discuss the performance impact of various other implementation details in the flyweight pattern. In section 4.2, we present a concluding performance evaluation over a wide variety of test cases generated by various frontends.

4.1 Performance of Different Data Structures

The table below shows the performance change of the flyweight implementation using different map data structures for the flyweight pool implementation which stores term instances. The performance change is relative to the base implementation without flyweight pattern.

Data Structure	Relative Performance Change (Negative is better)
`mutable.HashMap`	-1.3%
`mutable.WeakHashMap`	-0.2%
`concurrent.TrieMap`	-0.2%
`concurrent.ListMap`	+89.5%

As expected, the use of `ListMap` significantly worsens performance, as linear time with respect to existing terms is required for a lookup operation. The performance of `HashMap`, `WeakHashMap` and `TrieMap` are very similar to the base implementation in this benchmark. As Silicon may use multiple verifier instances in parallel, we chose `TrieMap` for the concluding performance evaluation in section 4.2, as it has the additional benefit of being concurrency-safe.

4.1.1 Caching Libraries

Dedicated maps for caching such as Caffeine [4] where tested as well, but they add no advantage over maps implemented in the Scala standard library, performance- or otherwise. Eviction policies implemented in such caching libraries for example add an additional performance overhead, but are uncessary when used in our flyweight pool as terms are required to stay in the pool at least as long as other references to the term still exist.

4.1.2 Clearing Pools After Each File

As an attempt to increase performance, we modified the term pool discussed in 4.1 to be emptied after the verification of each file. However no significant performance difference could be observed.

4.2 Concluding Performance Evaluation

To measure the performance difference resulting from the flyweight pattern, programs generated by the VerCors, Prusti, Gobra and Vyper frontends are considered.

TODO: More in detail

Silicon optionally allows parallel verification, however the number of parallel verifiers is set to one. As Scala's `mutable.TrieMap` is used, the flyweight pattern would still work in a parallelized environment.

The benchmark is repeated ten times, where the slowest and fastest verification time are ignored. The flyweight implementation 2% was slower on average, which is still within the standard derivation of 2.9%.

4.3 Why Did Flyweight Fail

Although reference equality checks are certainly much faster than recursive structural equality checks, changing terms to use the Flyweight pattern didn't result in measurable performance improvements.

There are some reasons why this might be the case. First, there may be not enough structurally equal term instances to justify a flyweight pattern. To explore this possibility, we measured the hit percentage of looking up terms in the flyweight pool. Many structurally different term instances would lead to a low hit percentage, which would render a flyweight pattern inefficient. In our benchmarks, a hit percentage of around 83% was measured, meaning that on average, for every term created and added to the flyweight pool, four structurally equal terms could be avoided.

Another reason may lie in the depth of the terms on an equality check. If the terms are very flat when checking for equality, the additional performance overhead of structural, recursive equality checks becomes negligible compared to reference equality checks, even if many equality checks take place. To support this hypothesis, we counted the number of subterms contained term at every equality check. For example, if `Plus(1, Minus(2, 3))` was checked for equality, we count 5 contained subterms: once `Plus`, once `Minus` and three integer literals. Figure 4 shows the average subterm count for each term class. On average, a term contains around 14 subterms on equality check.

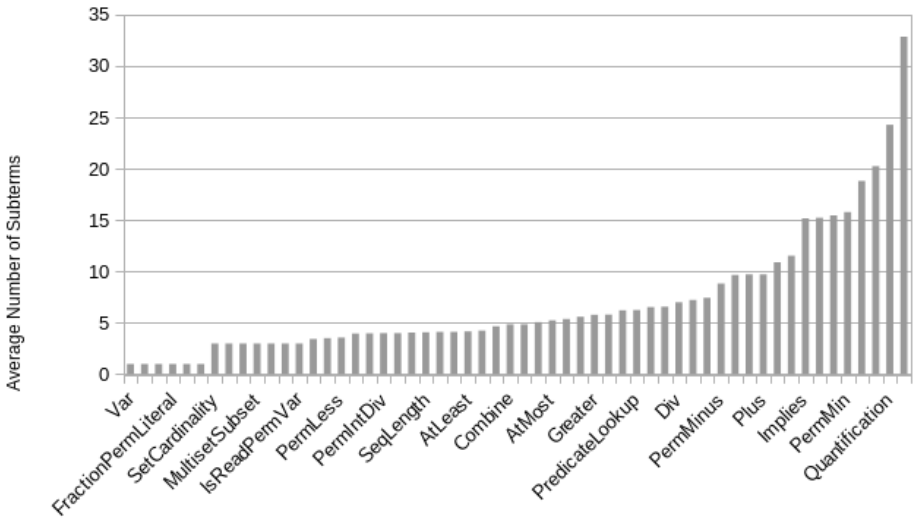


Figure 4: Average number of subterms contained in a term instance on equality check.

To summarize, avoiding on average four structurally equal term instances

which contain on average 14 subterms is most likely not enough to justify the overhead introduced by the flyweight pattern.

4.4 Using Macros to Facilitate Experiments

Although the flyweight pattern itself didn't have a significant impact on performance, the macro annotation developed to implement the flyweight pattern can be quickly modified to perform experiments or benchmarks on the Silicon AST.

In the following example, the macro is edited to ignore AST simplifications. The method `_apply` performs AST simplifications. To ignore them, we can avoid calling `apply` and instead, we directly create instances using the `new` keyword. Using the macro, this can be done quickly for all terms by only modifying three lines instead of rewriting every term manually.

Listing 4: Use AST simplifications as normal.

```
def apply(..$fields) = {
    // ...
    ${
        if (hasRenamedApplyMethod)
            // AST simplifications are potentially performed when
            // creating instances via _apply.
            q"${termName}._apply(..${fieldNames})"
        else
            q"new $className(..${fieldNames})"
    }
    // ...
}
```

Listing 5: Modified macro to ignore AST simplifications.

```
def apply(..$fields) = {
    // ...
    ${
        q"new $className(..${fieldNames})"
    }
    // ...
}
```

This illustrates that the macro developed as part of this thesis facilitates experiments and benchmarks in Silicon.

4.5 Complementary Benchmarks

4.5.1 Parallelization

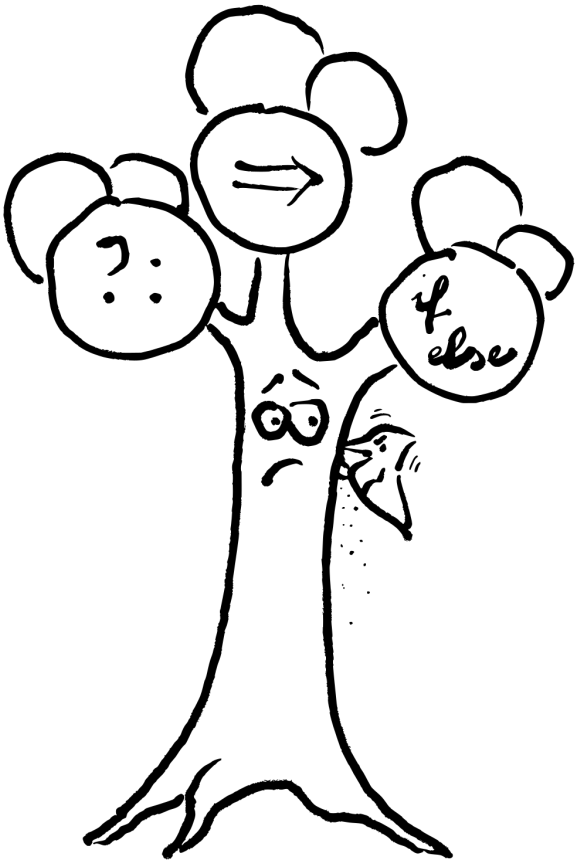
5 Future Work

- The macro annotation developed to modify Silicon's Terms invites to various experiments. AST simplifications for the Silicon AST are concurrently hard-coded in the corresponding `apply` methods of the terms. `Plus(t, IntLiteral(0))` for example is directly simplified to `t`. The usage of a domain-specific language in combination with Scala macros to auto-generate AST simplifications would allow easy addition and modification of AST simplifications in the future.

- In principle, the generic plugin implementation introduced in section 3.3.1 can be used in projects other than Silicon which make use of their own macro annotations. however, the plugin is not yet fully fleshed out. Concretely, the plugin slows down as it reads the many configuration files generated by the macro. This could be avoided by using caching techniques, for example. Furthermore, the plugin does not work yet for all kind of macro annotations. It can be developed further to be faster and provide more support for a wider range of macro types, providing a valuable addition to development of Scala programs using macros in the IntelliJ IDE.

Part II

Joining;
Getting Rid of the Branches



6 Approach

6.1 Where to Join

For verifying a program, Silicon uses the symbolic execution approach, where the program is interpreted, and a symbolic state keeps track of all possible program states at the current point of execution, for all possible input values of the program. When encountering certain expressions or statements, for example an if-statement, symbolic execution branches with the assumptions of the corresponding program path. In the following sections, we list the different statements and expressions that can be joined again.

6.1.1 Conditional Expressions

Consider a conditional expression of the form $Ite(b, e_1, e_2)$. As Silicon evaluates this expression, and b cannot definitely be evaluated to true or false, two branches are created, the first one assumes that b is true and the expression is evaluated to be e_1. the second branch assumes that b is false, and the evaluation yields e_2. The symbolic execution is continued in both branches.

6.1.2 Implications

Similar to conditional expressions, symbolic execution branches on implications too. Consider an implication of the form $b \implies i$. The first branch assumes that b is true and consequently the implication i holds. The second branch assumes that b is false and the implication i may or may not hold.

6.1.3 If-Statements

Viper is parsed into a Control Flow Graph (CFG) consisting of blocks containing statements, and edges which connect blocks. Edges can be unconditional or conditional, and can potentially form cycles whenever a back edge is connected to a loop head block. Symbolic execution branches whenever a block has more than one outgoing edges.

To join again at the correct location within the CFG after branching, the join point for each corresponding branch point has to be identified. We introduce a recursive algorithm which maps each branch point to its corresponding join point, if it exists:

1. Initialize a queue of blocks to visit and a list of already visited blocks. Traverse the CFG in a breath-first way.
2. *Base Case.* If a block is visited which already is included in the visited list, return this block as it is the join point corresponding to the branch point where this procedure was called.
3. *Recursive Case.* If a block has two outgoing edges, it is a branch point. Call this procedure recursively, starting from this branch point.

Listing 6: The join point finding algorithm without support for loops.

```
1  def findJoinPoint(branchPoint)
2      queue = branchPoint.successors
3      visited = []
4      while curr = queue.next
5          if curr not in visited
6              visited += curr
7              match curr.successors
8                  case [next]:
9                      queue += next
10                 // Branch point found,
11                 // find corresponding join point recursively.
12                 case [next1, next2]:
13                     queue += findJoinPoint(curr)
14         else
15             // curr is the join point for this branch point.
16             return curr
```

Special attention has to be paid to loops. If our algorithm follows a back edge before finding a join point, it may do the recursion again for the same branch point, leading to non-termination. To avoid this, all already visited loop head blocks are remembered for later recursive invocations. Already visited loop head blocks are not followed again. Whenever a join point

exists for a branch point created via if-statement, we are now able to join the branches.

Branches created by both conditional expressions and implications are already being joined if they are pure. Branches resulting from impure conditional expressions and implications, and from all if-statements however aren't joined again, meaning that all statements later down the verification path are evaluated twice. Both of these verification paths may branch again, eventually leading to exponential growth in branches. These branches are avoided when joining the symbolic execution paths back together. However, joining execution paths requires merging the symbolic state, which ultimately produces more complex symbolic state entries.

Intuitively, the same work has to be done with or without joining. The difference is that no joining leads to many execution paths with simpler symbolic states, and thus more in quantity, but less complex invocations of the SMT solver. More joining on the other hand leads to fewer execution paths but with more complex symbolic states, resulting in fewer, but more complex invocations of the SMT solver.

6.2 Merging the Symbolic State

After finding the appropriate locations for joining, the information gathered through both branches in the symbolic state have to be merged into a single symbolic state to continue symbolic execution in a single path.

To formalize the merging process, we define a symbolic state σ of type $\Sigma := (\Gamma, \Pi, H)$. The entries defined as follows:

- A store γ of type $\Gamma := Var \to V$ maps local variables to their symbolic values.
- A path condition stack π of type Π records all assumptions that have been made in the current verification path.
- A symbolic heap h of type H that records which locations are accessible and their respective symbolic values.

For the following subsections, assume that after the verification branched un-

der the condition c of type $Bool$, two symbolic states $\sigma_1 = (\gamma_1, \pi_1, h_1)$ under the branch condition $c = c_1$, and $\sigma_2 = (\gamma_2, \pi_2, h_2)$ under the branch condition $\overline{c} = c_2$ are to be merged, resulting in the new state $\sigma_3 = (\gamma_3, \pi_3, h_3)$.

Note that this core idea could be extended to merge more than two states at once. In practice however, no more than two states are merged at once.

6.2.1 Merging the Store

For merging stores γ_1 and γ_2, we consider two cases:

1. For some local variable x, we have $x \mapsto v_1 \in \gamma_1$ and $x \mapsto v_2 \notin \gamma_2$. In this case, we can simply omit x in the new store γ_3 as we can assume that x won't be needed later down the verification path.
2. For some local variable x, we have $x \mapsto v_1 \in \gamma_1$ and $x \mapsto v_2 \in \gamma_2$. In this case, we modify the heap chunk such that $x \mapsto Ite(c_1, v_1, v_2) \in \gamma_3$.

6.2.2 Merging the Heap

A heap is essentially a collection of heap chunks, where each heap chunk provides information about the location's value and the receiver's permission amount to the location. As there may be multiple heap chunks making statements about aliased receivers, Silicon provides a mechanism to merge them using a mechanism called state consolidation. To merge heaps h_1 and h_2, we perform the following steps:

1. Every non-quantified heap chunk c for which $c \in h_1$ and $c \in h_2$ holds can be carried over to h_3 without modifications.
2. Non-quantified heap chunks $c := x.f \mapsto t \# p$ where $c \in h_1$ and $c \notin h_2$ are modified to have permissions only if c_1 holds: $c' := x.f \mapsto t \# Ite(c_1, p, 0) \in h_3$
3. Finally, h_3 can be consolidated to avoid multiple aliasing heap chunks.

Quantified heap chunks are of the shape $\forall x : c(x) \implies e(x).f \mapsto v(x) \# p(x)$. In practice, they are rewritten to the equivalent form $\forall x : e(x).f \mapsto v(x) \# p'(x)$, where $p'(x)$ is defined as:

$$p'(x) = \begin{cases} p(x) & \text{if } c(x) \text{ is true,} \\ 0 & \text{else} \end{cases}$$

Analogously to non-quantified heap chunks, we can simply modify the permission amounts of quantified heap chunks to $p''(x) = Ite(c_1, p'(x), 0)$.

6.2.3 Merging the Path Conditions

For path conditions, the functionality for merging is already provided. This is done by putting the collected path conditions of each branch under an implication with the corresponding branch condition.

7 Implementation

7.1 Finding Join Points

To find join points within the CFG, the algorithm described in listing 6 is implemented, but with some additional modifications to support loops. All loop heads that were already seen are remembered to avoid infinite recursion issues as described in 6.1.3. The algorithm produces a mapping from each branch point to the respective join point. If an if-statement is encountered during verification, we check if a corresponding join point exists and the branches can be joined again.

7.2 Implementing State Merges

Store and heap merges are implemented as according to section 6.2. Silicon's state however consists of some more fields carrying additional data, that have to be merged with caution. Caches within the state are emptied instead of merged for simplicity.

Our implementation can be optionally enabled by passing the command-line argument `--moreJoins` to the Silicon executable. To illustrate the difference in verification, consider the Viper program in listing 7. Table 1 shows the execution trace of Silicon with more joins disabled, in table 2, more joins is enabled.

Listing 7: An example Viper program.

```
1  method test(b: Bool) {
2      var x: Int := 0
3      if (b) {
4          x := x + 5
5      } else {
6          x := x + 7
7      }
8      x := x + 3
9      assert x <= 10
10 }
```

Operation	Store	Path Conditions
`var x: Int := 0`		
`x := x + 5`	$x \mapsto 0$	b
`x := x + 3`	$x \mapsto 5$	b
`assert x ⩽ 10`	$x \mapsto 8$	b
`x := x + 7`	$x \mapsto 0$	$!b$
`x := x + 3`	$x \mapsto 7$	$!b$
`assert x ⩽ 10`	$x \mapsto 10$	$!b$

Table 1: Symbolic execution trace of viper program 7 using the base implementation without joining.

Operation	Store	Path Conditions
`var x: Int := 0`		
`x := x + 5`	$x \mapsto 0$	b
`x := x + 7`	$x \mapsto 0$	$!b$
`x := x + 3`	$x \mapsto Ite(b, 5, 7)$	
`assert x ⩽ 10`	$x \mapsto Ite(b, 5, 7) + 3$	

Table 2: Symbolic execution trace with joining. Joining leads to less execution steps, but to a more complex state.

8 Evaluation

In the following section 8.1, we analyze the performance difference of our implementation using more joins, compared to the base implementation which only joins on simple cases, as described 6. in Additionally, we provide some complementary benchmark results which were a byproduct of this thesis.

8.1 Concluding Performance Evaluation

Benchmark on various frontend-generated Viper programs shows that verification time using more joins decreases by around 3% on average, relative to a version which doesn't make use of the implemented joining procedures. Intuitively, one would expect that fewer branches lead to better performance, however state merging introduces a more complex final state which again tends to worsen performance.

When comparing the number of state merges that occur during verification to the verification time performance difference, no clear correlation is visible, as can be seen in figure 6.

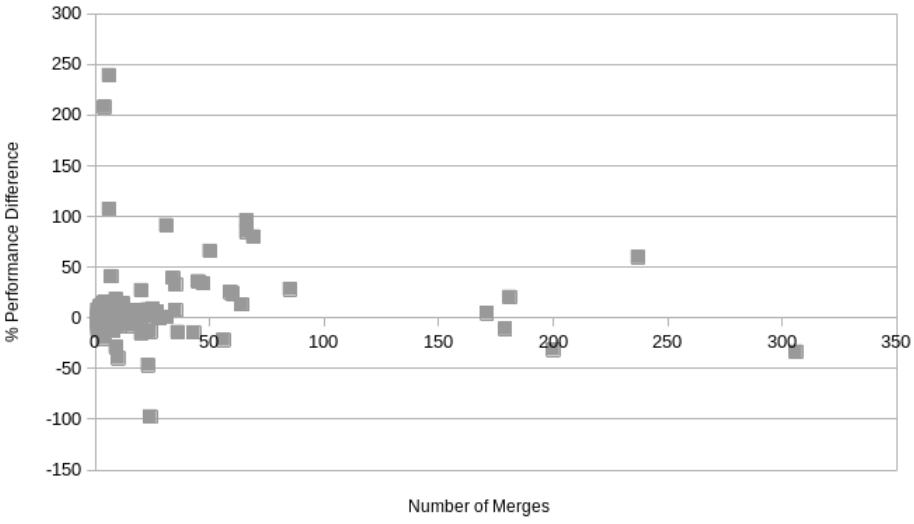


Figure 5: Impact on the number of state merges on the performance, negative performance difference shows a speedup.

Interestingly, the performance seems to improve about 3.3% for programs with an absolute base verification time up to 0.5 seconds. With increasing verification time, the performance decreases, and for programs with a verification time greater than 10 seconds, we get a decrease of performance of nearly 25%.

This observation suggests for smaller programs where fewer joins are needed, the more complex symbolic states caused by joining is worth trading for the benefit of having fewer branches. For larger programs, the symbolic state may become overly complex up to a point that the advantage of fewer branches no longer pays off.

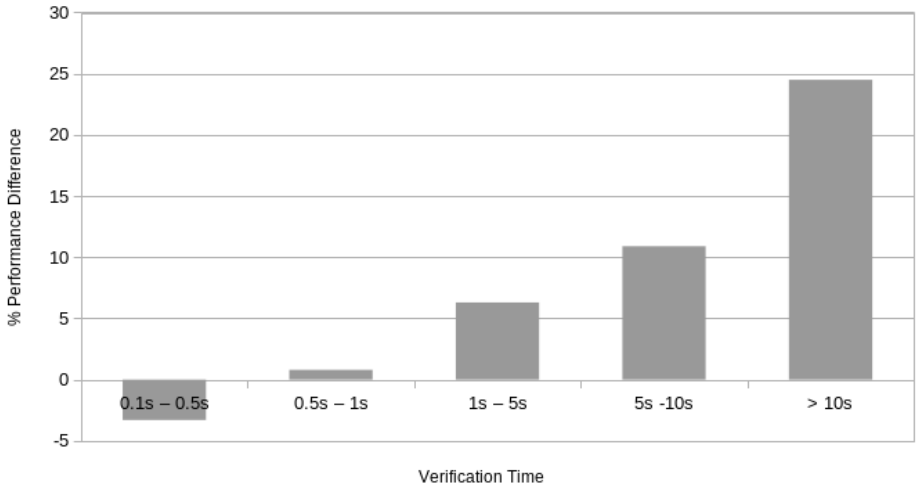


Figure 6: Change in performance depending on absolute base verification time. Negative performance difference shows a speedup.

8.2 Complementary Benchmarks

8.2.1 Disable Caching

As state merging currently empties caches instead of merging them, an option to disable the caches entirely was added to Silicon in order to provide more fair benchmarks. Disabling caches results in a performance decrease of 1.9% compared to the base implementation with caching enabled.

8.2.2 More Complete Exhale

Silicon additionally provides an option of enabling a more complete version of exhaling permissions, which must be used when joining is enabled. This is because joining may result in permissions of a location being divided into multiple heap chunks. Benchmarks have shown that enabling more complete exhale results in a performance increase of 2.9%.

9 Future Work

- Currently, we use conditional expressions for merging both the heap and the store. Merging can also be done by introducing new symbolic variables and conditionalizing them via implications in the path conditions. For example, if the store is merged to $v = Ite(b, e_1, e_2)$, we could analogously express this as $v = v'$ and restrict the value in the path conditions using implications $b \implies v' = e_2$ and $\overline{b} \implies v' = e_2$.

References

[1] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. "Modular Automatic Assertion Checking with Separation Logic". In: vol. 4111. Nov. 2005, pp. 115–137. ISBN: 978-3-540-36749-9. DOI: 10.1007/11804192_6.

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. en. 2016. ISBN: 978-0201633610.

[3] *IntelliJ API to Build Scala Macros Support*. https://blog.jetbrains.com/scala/2015/10/14/intellij-api-to-build-scala-macros-support/. Accessed: March 3, 2021.

[4] Ben Manes. *Caffeine GitHub Repository*. https://github.com/ben-manes/caffeine. Accessed: July 22, 2021.

[5] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A verification infrastructure for permission-based reasoning". en. In: *Dependable Software Systems Engineering*. Ed. by Alexander Pretschner, Doron Peled, and Thomas Hutzelmann. Vol. 50. Amsterdam: IOS Press BV, 2017, pp. 104–125. ISBN: 978-1-61499-809-9. DOI: 10.3233/978-1-61499-810-5-104.

[6] *Scala Docs - Macro Annotations*. https://docs.scala-lang.org/overviews/macros/annotations.html. Accessed: July 29, 2021.

[7] Malte H. Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". en. PhD thesis. Zürich: ETH Zurich, 2016. DOI: 10.3929/ethz-a-010835519.