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

September 6, 2021

Abstract

To advance program verification in practice, fast verification is crucial as it provides a more streamlined experience for developers. This thesis explores two different approaches 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 Silicon's AST structures. Applying the flyweight pattern avoids multiple instances of structurally equal AST 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.

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

Last but not least, I'd like to express my gratitude towards my family for providing me with a very pleasant home office environment.

Contents

1 Introduction

Viper [8] is a verification infrastructure on top of which verification tools for different programming languages can be built. Silicon [11] is a backend for Viper, which is based on Smallfoot-style [2] 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 [...]" [11]

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 called 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 [5] 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

In the following sections we discuss our approach of implementing the flyweight pattern on ASTs, and the advantage of using code generation via metaprogramming for our implementation.

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 [5] 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.

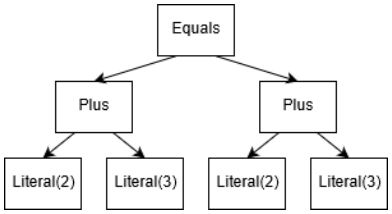


Figure 1: Example AST without using the flyweight pattern. Multiple structurally equal instances of the some term may exist in the AST.

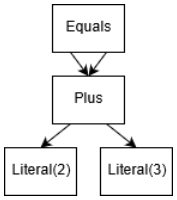


Figure 2: The same AST as in figure 1, but using the flyweight pattern. The two structurally equal term instances in figure 1 are avoided, and only one instance exists instead.

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 introduce 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. [10] 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 readability 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

After discussing the general idea of flyweight ASTs, we now focus on the specific implementation details for Silicon's ASTs. As the flyweight pattern for Silicon's terms is implemented in Scala, we assume general familiarity with the language.

3.1 Implementation of the Flyweight Pattern

For the implementation of flyweight ASTs in Silicon, we modify the AST terms as follows:

1. The constructor of a term is made private (listing 1, line 1) so that new term instances can't be created via the `new` keyword, but only via the `apply` method (listing 1, 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 (listing 1, line 7). This allows us to later look up whether a structurally equal term was created already (listing 1, line 10).
3. In the `apply` method, we check the pool for structurally equal instances (listing 1, line 10), and if one exists, we return it and thus avoid creating a new instance of the same term (listing 1, 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 (listing 1, 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

For a nice programming experience using Scala 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." [6]

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 3.

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 4.

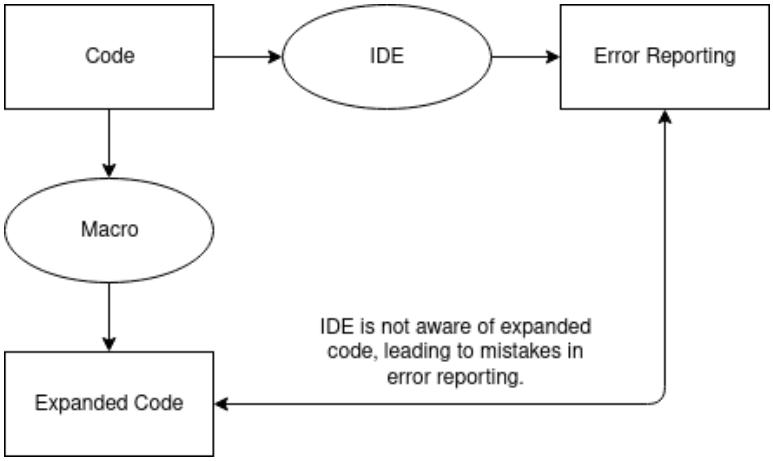


Figure 3: The IDE is not aware of code changes done by our macro. This leads to incorrect error reporting.

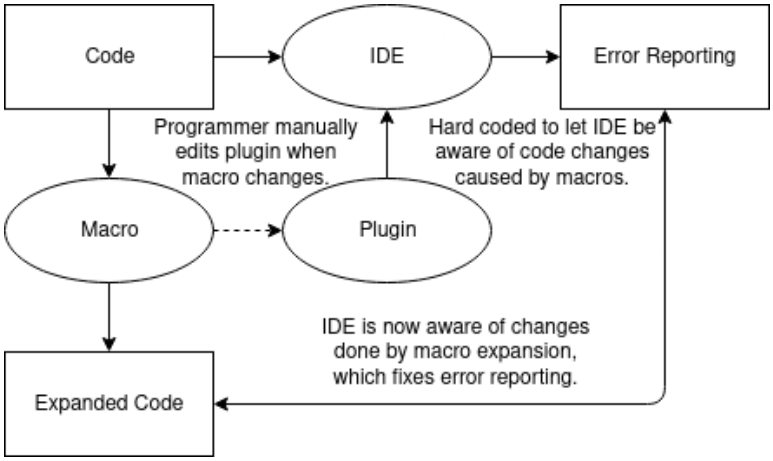


Figure 4: The IDE is now aware of code changes done by the flyweight macro, which fixes error reporting.

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 an XML configuration file whenever it is run by the compiler. This configuration file is then read by the IntelliJ plugin. The plugin communicates the method signatures read from the configuration file to the IDE, which now has all information to provide proper coding assistance. 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, without the programmer being required do manually rewrite the plugin. This is illustrated in figure 5.

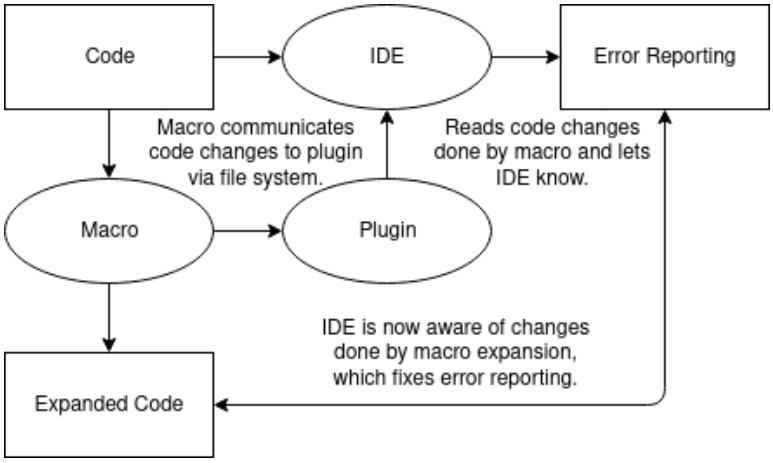


Figure 5: The IDE is aware of code changes done by the flyweight macro, which fixes error reporting. The plugin does not manually need to be edited whenever changes in the macro are made.

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 that stores term instances. The performance change is relative to the base implementation without flyweight pattern.

Data Structure	Relative Performance Change to Base Implementation (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 [7] 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 unnecessary 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 analyze the performance difference resulting from the flyweight pattern, test programs generated by the VerCors [3], Prusti [1], Gobra [13] and Vyper [12] frontends are used as we are interested in the performance impact on the verification of real-life examples. For the benchmark, total verification time is measured. The benchmark is repeated ten times, where the slowest and fastest verification times are ignored.

Silicon optionally allows multithreaded verification, but for this benchmark, multithreading is disabled. However, as Scala's `mutable.TrieMap` is used, the flyweight pattern would still work in a parallelized environment.

Figure 6 suggests that there is a small performance improvement of programs with an absolute verification time greater than ten seconds, but for programs with less than ten seconds absolute verification time, we observe a performance decrease. On average, the flyweight implementation 2% was slower, which is still within the standard derivation of 2.9%.

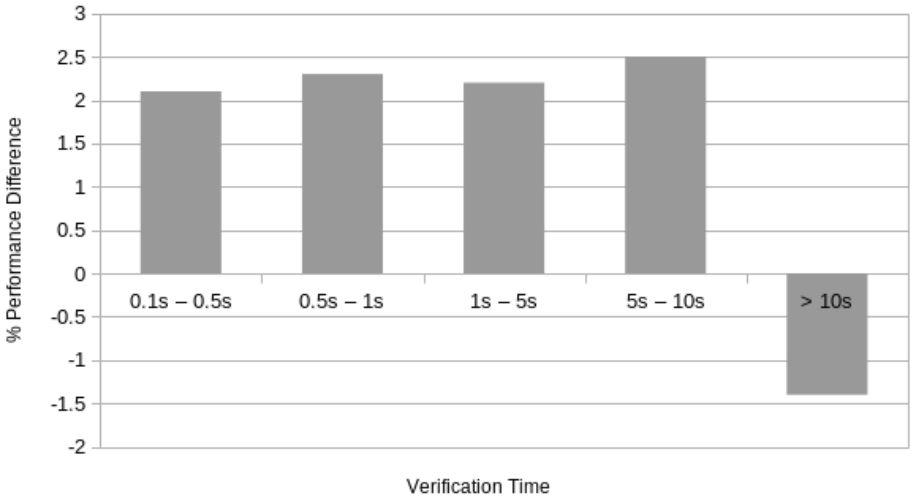


Figure 6: Change in performance depending on absolute base verification time. Negative performance difference shows a speedup.

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. Remember that on the creation for every term, we first check if a term instance with the same component already exists in the term pool (listing 1, line 10). If a term already exists in the pool, we call it a hit, else a miss. 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, only four structurally equal term instances 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 perfor-

mance overhead of structural, recursive equality checks becomes negligible compared to reference equality checks, even if many equality checks take place. To test 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 with some other term, we count 5 contained subterms: once `Plus`, once `Minus` and three integer literals. Figure 7 shows the average subterm count for each term class. On average, a term contains around 14 subterms on equality check.

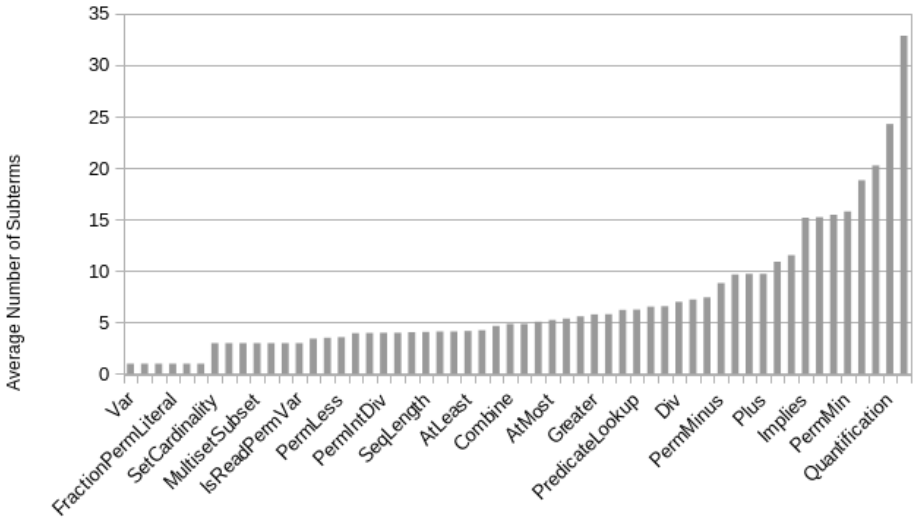


Figure 7: Average number of subterms contained in a term instance on equality check.

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 Complementary Benchmarks

4.4.1 Parallelization

In this section we discuss the impact of parallelization in Silicon. Figure 8 illustrates the change of performance if parallelization is enabled for the

base implementation using 8 threads, relative to the base implementation without parallelization.

For small programs, the overhead introduced by parallelization isn't worth the speedup, and performance decreases up to 100%, which isn't much in absolute terms as absolute verification times are quite small. For larger programs with absolute verification time of five seconds and above, parallelization provides a clear performance improvement of up to around 50%.

Figure 9 shows that the implementation using the flyweight pattern has no impact on parallelization, as expected.

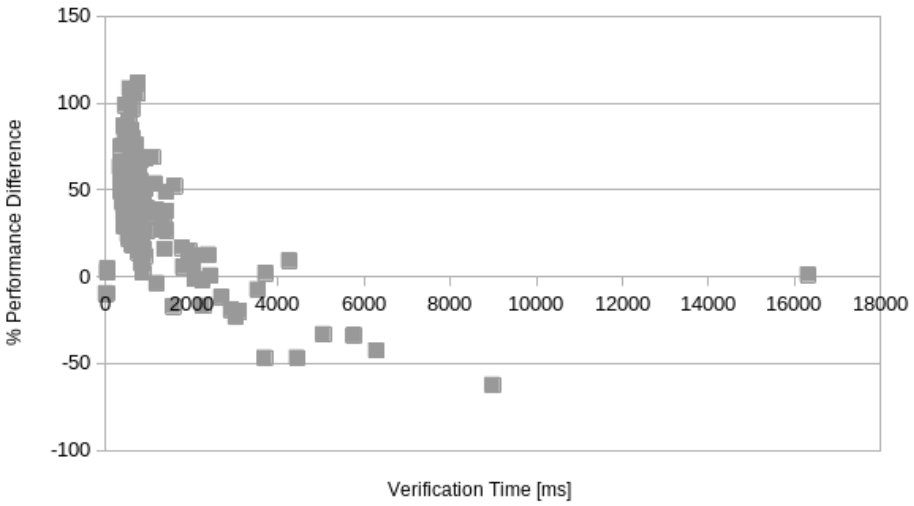


Figure 8: Change in performance when parallelization is enabled in the base implementation, compared to the base implementation without parallelization. Negative performance difference shows a speedup.

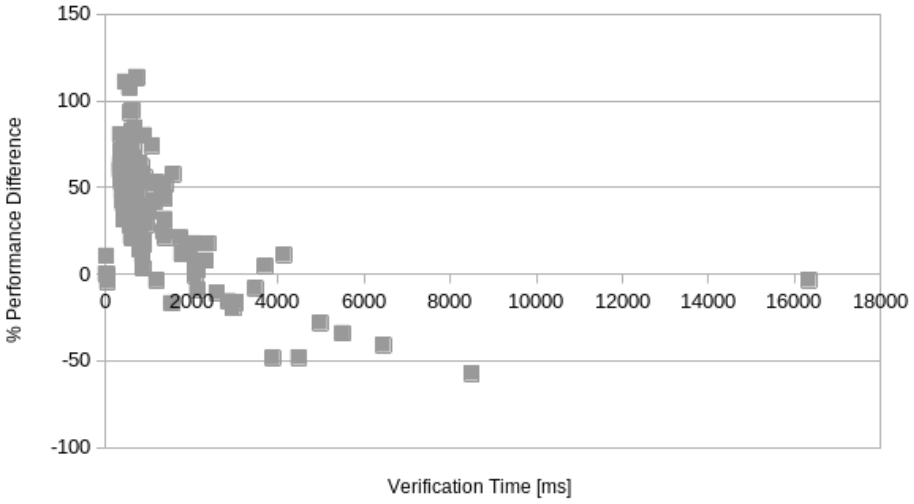


Figure 9: Change in performance when parallelization is enabled in the flyweight implementation, compared to the base implementation without parallelization. Negative performance difference shows a speedup.

4.5 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 don't call `apply` and instead, we directly create instances using the `new` keyword. As the AST simplifications take place in the `apply` method, they are now circumvented. 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.

```
1 def apply(..$fields) = {
2     // ...
3     ${
4         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.

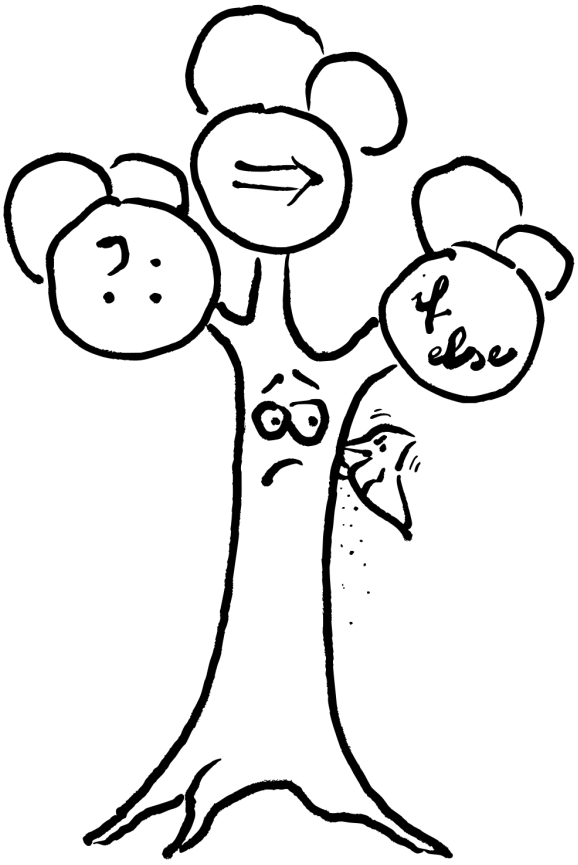
5 Conclusion and Future Work

In this part, we have introduced the concept of flyweight ASTs, which avoids multiple structurally equal terms within the same AST. This has the advantage of being able to replace structural, recursive equality checks by simple reference equality checks, but also adds additional overhead due to the flyweight pattern. The benchmark has shown that for most programs, a small performance decrease is visible when using flyweight ASTs. However, the macro annotation and IDE plugin developed as a part of this thesis invite for further experimentation in other directions:

- 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;
Reducing Verification Branches



6 Approach

6.1 Where to Join

For verifying a program, Silicon symbolically executes the program, 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 into two execution paths, where in one path the if-statement is assumed to evaluate to true, and in the second path to false. In the following sections, we list the different statements and expressions that result in a branching of execution paths, which later can be joined again.

Listing 6: An example Viper program to demonstrate how branching relates to the number of symbolic execution paths. Let `s0`, `s1`, `s2` and `s3` denote generic viper statements.

```
1 method test(b: Bool) {
2     s0
3     if (b) {
4         s1
5     } else {
6         s2
7     }
8     s3
9 }
```

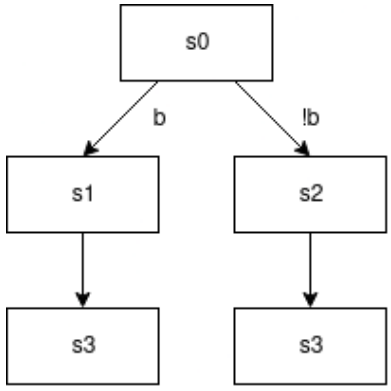


Figure 10: The symbolic execution paths taken by Silicon when verifying the program in listing 6. Note that `s3` is symbolically executed twice as the paths are not joined after the if-statement.

6.1.1 If-Statements

Viper is parsed into a control flow graph (CFG) consisting of blocks containing statements, and edges that connect blocks. Edges can be unconditional or conditional, and can potentially form cycles whenever a back edge is connected to a loop head block. Silicon branches whenever a block has more than one outgoing edges. If-statements cause such CFG blocks with 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. The algorithm used to identify join points within the CFG is discussed later in 7.1

6.1.2 Conditional Expressions

Consider a conditional expression of the form b `?` e_1 `:` e_2. As Silicon evaluates this expression, and b cannot definitely be evaluated to true or false, two branches are created, in the first one, b is assumed to be true, and the expression is evaluated to be e_1. In the second branch, b is assumed to be false, and the evaluation yields e_2. The symbolic execution is continued in both branches, and all following code is executed twice.

6.1.3 Implications

Similar to conditional expressions, symbolic execution branches on implications too. Consider an implication of the form $b_1 \implies b_2$. Again, in the first branch, b_1 is assumed to be true, and in the second branch, b_1 is assumed false.

6.2 Pros and Cons of Joining

Branches created by conditional expressions and implications are already being joined if they are pure, that is, they do not modify the heap. Branches resulting from impure conditional expressions and implications, and from all if-statements aren't joined again, meaning that all statements later down the verification path are evaluated twice, as demonstrated in the example of figure 10. 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.

Intuitively, the same work has to be done with or without joining, but there are some differences when joining. Concretely, joining leads to fewer execution paths but with more complex symbolic states. Table 1 lists some differences.

Property	Without Joining	With Joining
Number of Execution Paths	More Paths	Fewer Paths
Symbolic State	Less Complex	More Complex
Number of SMT Solver Invocations	More Invocations	Fewer Invocations
Complexity of SMT Solver Invocations	More Complex	Less Complex

Table 1: Differences of using symbolic execution with joining versus without joining.

6.3 Merging the Symbolic State

After finding the appropriate locations for joining, the information gathered through both branches in the symbolic state has to be merged into a single

symbolic state to continue the symbolic execution on 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 on the current verification path.
- A symbolic heap h of type H that records which heap locations are accessible and their respective symbolic values. A heap is implemented as 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.

For the following subsections, assume that after the verification branched under the condition c of type $Bool$, two symbolic states $\sigma_1 = (\gamma_1, \pi_1, h_1)$ under the branch condition b_1, and $\sigma_2 = (\gamma_2, \pi_2, h_2)$ under the branch condition b_2, which is the negation of b_1, 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 the current version of Viper however, no more than two states are merged at once.

6.3.1 Merging Stores

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

1. Let $x \mapsto v$ denote that the variable x maps to the value v. Assume that 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 as Viper has the usual lexical variable scopes.
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 store entry such that $x \mapsto Ite(b_1, v_1, v_2) \in \gamma_3$,

where $Ite(b, e_1, e_2)$ is a conditional expression that resolves to e_1 if b is true, and e_2 otherwise.

6.3.2 Merging Heaps

To merge heaps h_1 and h_2, we perform the following steps:

1. Every 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. Let $r.f \mapsto v \# p$ denote a heap chunk where r, v, p are symbolic expressions denoting the receiver of some location f, the symbolic value of the location and the permission amount provided by the heap chunk. Heap chunks $c := r.f \mapsto t \# p$ where $c \in h_1$ and $c \notin h_2$ are modified to have permissions only if b_1 holds: $c' := r.f \mapsto v \# Ite(b_1, p, 0) \in h_3$.

Quantified heap chunks are of the shape $\forall r : r.f \mapsto v(r) \# p(r)$. Analogously to non-quantified heap chunks described previously, we can simply modify the permission amounts of quantified heap chunks to $p'(x) = Ite(b_1, p(x), 0)$.

Proving assertions or permission amounts of a location in the heap is normally done greedily, where the heap chunks are traversed until the first matching heap chunk for the location is found. This greedy algorithm is in general incomplete if multiple heap chunks for the same location exist. To avoid any issues, Silicon's option `--enableMoreCompleteExhale` should be used, which enables a more sophisticated method of finding matching heap chunks. Furthermore, more complete exhale provides a small performance improvement by itself, as later discussed in section 8.2.2.

Another option to avoid incompleteness is to do state consolidations after merging heaps. State consolidations can rewrite the heap using the information available in the current state such that aliasing heap chunks can be combined into a single one. Doing state consolidations after every merge has shown to decrease performance, which is why we require more complete exhale to be enabled instead.

6.3.3 Merging 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.

6.3.4 An Example

Assume we want to merge two following two symbolic states:

$\sigma_1 = (\{a \mapsto 7\}, \{b <= 10\}, \{x.f \mapsto 7 \# 1\})$ with branch condition $b <= 10$
$\sigma_2 = (\{a \mapsto 8\}, \{b > 10\}, \{y.f \mapsto 8 \# 1\})$ with branch condition $b > 10$

As a occurs in the store of both σ_1 and σ_2, the new store entry has the shape $a \mapsto Ite(b <= 10, 7, 8)$.

For the heap, we cannot be sure whether the receivers x and y are aliases, which is why we keep both heap chunks but with modified permission amounts: $x.f \mapsto 7 \# Ite(b <= 10, 1, 0)$, $y.f \mapsto 8 \# Ite(b > 10, 1, 0)$.

Finally, the path conditions $b <= 10$, $b > 10$ are merged, resulting in a new, empty path condition.

The new state is now fully described as follows:

$\sigma_3 = (\{x \mapsto Ite(b <= 10, 7, 8)\}, \{\}, \{x.f \mapsto 7 \# Ite(b <= 10, 1, 0), y.f \mapsto 8 \# Ite(b > 10, 1, 0)\})$

7 Implementation

7.1 Finding Join Points

In section 6.1, we have discussed which statements and expressions cause branches that can be joined again, but we still need to identify the join points. For conditional expressions and implications, finding the join point is trivial as it is within the same expression as the branch point. For if-statements, finding the join point to a corresponding branch point is more difficult as we have to search for it within the whole CFG. We introduce a recursive algorithm which maps each branch point to its corresponding join point, if it exists. The algorithm runs the following steps:

1. Initialize a queue of blocks to visit next (the successors of the branch point), and a list of already visited blocks (the branch point itself). Traverse the CFG in a breath-first way by adding the successors of a block to the queue (listing 7, line 9).
2. *Base Case.* If a block is visited that already is included in the visited list, return this block (listing 7, line 16), as it is the join point corresponding to the branch point where this algorithm was called.
3. *Recursive Case.* If a block has two outgoing edges, it is a branch point. Call this algorithm recursively, starting from this branch point (listing 7, line 13).

Listing 7: The join point finding algorithm without support for loops.

```
1  def findJoinPoint(branchPoint: CFGNode)
2      queue = branchPoint.successors
3      visited = [branchPoint]
4      // Abort while the loop if no next node is given.
5      while curr = queue.next
6          if curr not in visited
7              visited += curr
8              match curr.successors
9                  case [next]:
10                     queue += next
11                 // Branch point found,
```

```
12                  // find corresponding join point recursively.
13                  case [_, _]:
14                      queue += findJoinPoint(curr)
15                  // Currently, only CFGs with at most two outgoing
16                  // edges are considered.
17                  case _:
18                      abort("At most two outgoing edges are supported")
19
20          else
21              // curr is the join point for this branch point.
22              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. Figure 11 shows an example CFG where this problem occurs. To avoid this issue, all already visited loop head blocks (they are already labeled as such) are remembered for later recursive invocations. Already visited loop head blocks are not followed again.

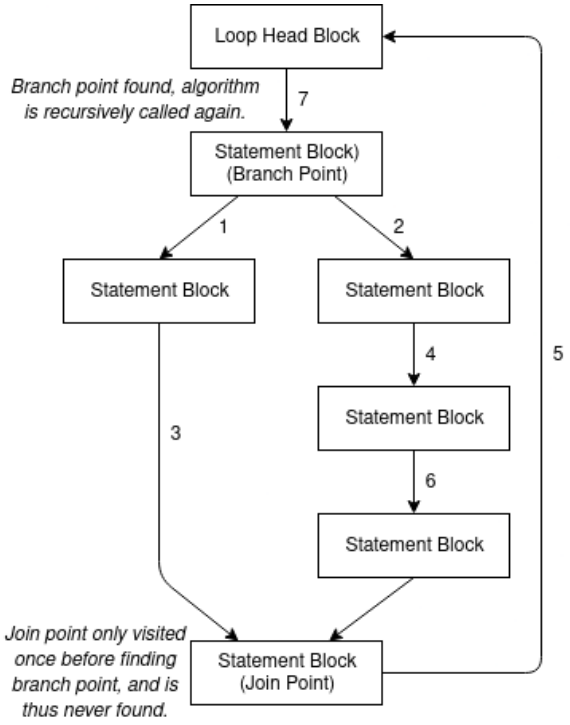


Figure 11: An example where the naive join point finding algorithm sketched in listing 7 fails. Starting at the branch point, the edges are visited in a breath-first way, as indicated by the numbers. Due to the loop, the initial branch point is found again before finding the join point, and the same recursion is done again.

To find join points within the CFG, the algorithm described in listing 7 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 figure 11. In a single preprocessing step, 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. Otherwise, we branch as normal without joining again.

7.2 Implementing State Merges

Store and heap merges are implemented as according to section 6.3. Silicon's state, however, consists of additional data, that has to be merged with caution. Some of these additional state components are used at various times during verification to increase performance. Such caches within the state are currently emptied instead of merged for simplicity. Furthermore, the option `--disableCaches`, which disables the caches, was added to Silicon in order to provide more fair benchmarks. The performance impact of disabling caches is further discussed in section 8.2.1.

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 8. Table 2 shows the execution trace of Silicon with more joins disabled, in table 3, more joins is enabled.

Listing 8: 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 }
```

Line (Listing 8)	Operation	Store	Path Conditions
2	`var x: Int := 0`		
4	`x := x + 5`	$x \mapsto 0$	b
8	`x := x + 3`	$x \mapsto 5$	b
9	`assert x ⩽ 10`	$x \mapsto 8$	b
6	`x := x + 7`	$x \mapsto 0$	$!b$
8	`x := x + 3`	$x \mapsto 7$	$!b$
9	`assert x ⩽ 10`	$x \mapsto 10$	$!b$

Table 2: Symbolic execution trace of viper program 8 using the base implementation without joining.

Line (Listing 8)	Operation	Store	Path Conditions
2	`var x: Int := 0`		
4	`x := x + 5`	$x \mapsto 0$	b
6	`x := x + 7`	$x \mapsto 0$	$!b$
8	`x := x + 3`	$x \mapsto Ite(b, 5, 7)$	
9	`assert x ⩽ 10`	$x \mapsto Ite(b, 5, 7) + 3$	

Table 3: Symbolic execution trace with joining. Joining leads to fewer 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. Additionally, we provide some complementary benchmark results which provide additional insights.

8.1 Performance Evaluation

The benchmark uses frontend-generated Viper programs from VerCors [3], Prusti [1], Gobra [13], Vyper [12] and Nagini [4] as we are interested in the performance impact on verifying real-life examples, and the total duration of each verification run is measured. The benchmark is repeated five times, where the slowest and fastest verification times are ignored.

The results show that verification time increases by around 3% on average when using more joins, 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.

Interestingly, figure 12 shows that the performance seems to improve for about 3.3% of the programs with an absolute base verification time of up to 0.5 seconds. With increasing verification time, the performance seems to decrease.

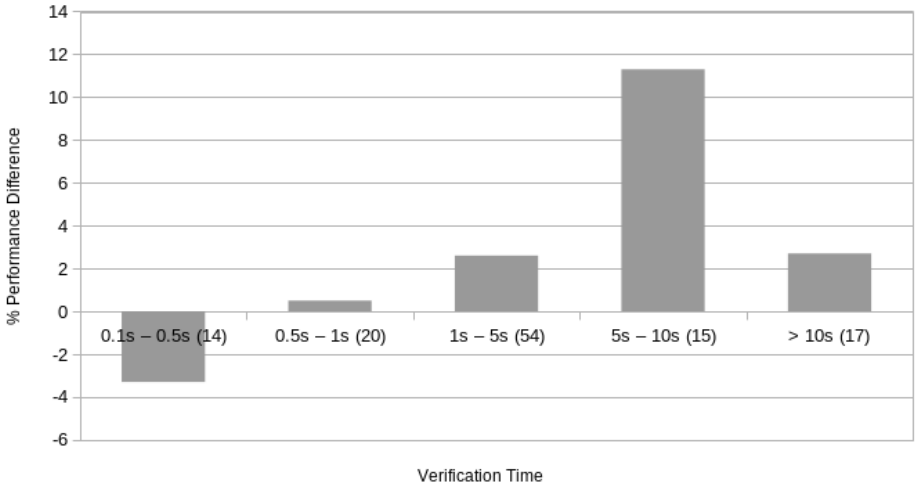


Figure 12: Change in performance depending on absolute base verification time. Negative performance difference shows a speedup. Excluded are programs where no joins were performed on verification. The numbers in the brackets indicate the number of test cases falling in each section.

This observation suggests that 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.

When comparing the number of state merges to the verification time performance difference, no clear correlation is visible, as can be seen in figure 13.

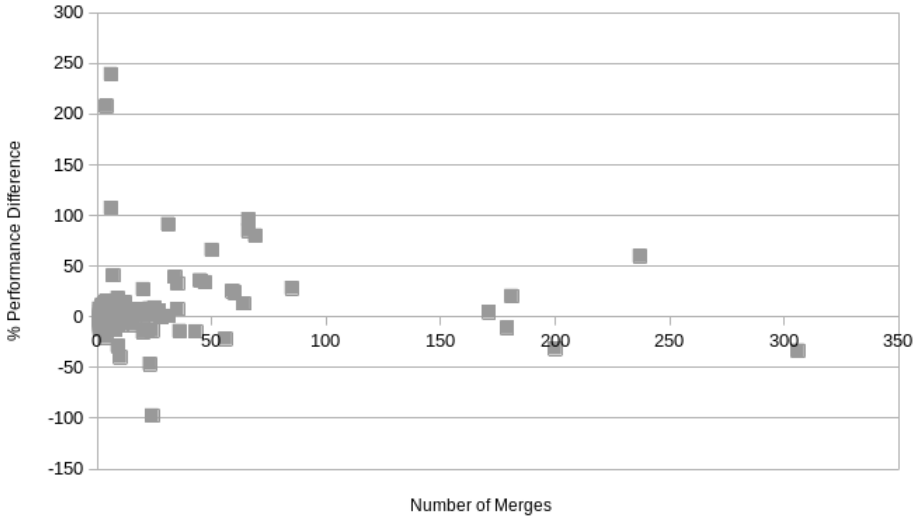


Figure 13: Impact on the number of state merges on the performance, negative performance difference shows a speedup.

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. In the base implementation used for comparison in section 8.1, the caches were disabled such that the performance of both implementations are not affected by caching.

8.2.2 More Complete Exhale

Silicon additionally provides an option of enabling a more complete version of exhaling permissions, which should be used when joining is enabled, as discussed in section 6.3.2. Benchmarks have shown that enabling more complete exhale results in a performance increase of 2.9%. In the benchmark

presented in section 8.1, more complete exhale was used by both implementations such that this performance increase observed by using this feature does not affect the results.

9 Conclusion and Future Work

We have presented an approach for joining verification branches in Silicon, and discussed the advantages and disadvantages of joining. The benchmark has shown that joining does not generally lead to a performance improvement when verifying programs. Nevertheless, there are still some open questions that can be addressed in 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 using a new symbolic variable v' as $v = v'$ and restrict the value of v' in the path conditions using implications $b \implies v' = e_2$ and $\overline{b} \implies v' = e_2$. Maybe, The implementation of merges using implications instead of conditional expressions will lead to a performance increase.
- Section 7.2 discusses how caches are used in Silicons state. For simplicity, we empty such caches instead of merging them. Implementing sophisticated cache merges would probably result in a performance increase when using more joins.
- The current implementation described in part II has introduced a bug which in a few cases (7 of the 333 tests used in the benchmark were affected) leads to a failure in the interaction with the SMT solver Z3 [9] that Silicon uses internally. Simply put, the merging of some quantified heap chunks may result in a Z3 warning because two triggers, that for themselves work flawlessly, may result in a new, invalid trigger when merged. The exact details or an approach to fix this bug are currently unclear and should be further examined.

References

[1] V. Astrauskas et al. “Leveraging Rust Types for Modular Specification and Verification”. In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 3. OOPSLA. ACM, 2019, 147:1–147:30. DOI: 10.1145/3360573.

[2] 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.

[3] Stefan Blom and Marieke Huisman. “The VerCors Tool for Verification of Concurrent Programs”. In: *FM 2014: Formal Methods*. Ed. by Cliff Jones, Pekka Pihlajasaari, and Jun Sun. Cham: Springer International Publishing, 2014, pp. 127–131. ISBN: 978-3-319-06410-9.

[4] Marco Eilers and Peter Müller. “Nagini: A Static Verifier for Python: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I”. In: July 2018, pp. 596–603. ISBN: 978-3-319-96144-6. DOI: 10.1007/978-3-319-96145-3_33.

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

[6] *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.

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

[8] 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.

[9] Microsoft Research. *Z3 Theorem Prover*. https://github.com/Z3Prover/z3. Accessed: August 26, 2021.

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

[11] 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.

[12] *Vyper*. https://github.com/viperproject/2vyper. Accessed: August 4, 2021.

[13] Felix A. Wolf et al. “Gobra: Modular Specification and Verification of Go Programs”. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17. URL: https://doi.org/10.1007/978-3-030-81685-8%5C_17.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Performance Improvements of a Program Verifier

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Bösiger	Fabian

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Eich, 30.08.2021	F. Bösiger

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.