Improving Performance of the VerCors Program Verifier

Henk Mulder[1], Marieke Huisman$^{[0000-0003-4467-072X]1}$, and Sebastiaan Joosten$^{[0000-0002-6590-6220]2\star}$

[1] University of Twente
Enschede, the Netherlands
[2] Dartmouth College
Hanover NH, USA
hmulder88@gmail.com, m.huisman@utwente.nl,
Sebastiaan.Joosten@dartmouth.edu

Abstract. As program verification tools are becoming more powerful, and are used on larger programs, their performance also becomes more and more important. In this paper we investigate performance bottlenecks in the VerCors program verifier. Moreover, we also discuss two solutions to the identified performance bottlenecks: an improved encoding of arrays, as well as a technique to automatically generate trigger expressions to guide the underlying prover when reasoning about quantifiers. For both solutions we measure the effect on the performance. We see that the new encoding vastly reduces the verification time of certain programs, while other programs keep showing comparable times. This effect remains when moving to newer backends for VerCors.

This is the preprint version to the paper found at: https://doi.org/10.1007/978-3-030-64354-6_3

1 Introduction

Program verification has been an active research area for many years now [9]. In particular, many program verification tools have been developed for many years (e.g., KeY [1], VeriFast [10,11], and Dafny [13] have all been developed for more than 10 years). In this paper we focus on the VerCors verifier, which focuses in particular on the verification of concurrent and distributed software, and has been under development since 2011 [2,4,3,5]. It is being developed with the ultimate goal to make verification usable for developers that are not necessarily formal method experts, but to reach this goal, still substantial work is needed.

As program verification tools are becoming more powerful and are used to verify larger programs, their performance also becomes more and more a crucial factor. In particular if we want non-developers to be end-users of the verification tools, a good performance and reaction time is essential for acceptance of the

* Work done while at the University of Twente.

technology (in addition to many other aspects, such as the amount of specifications that have to be written, understandability of error messages etc.).

In this paper we discuss work that we did to improve the performance of VerCors. VerCors is developed as a front-end for Viper [16], which is an intermediate verification language framework that understands access permissions. VerCors encodes annotated programs written in a high-level programming language into Viper; Viper then uses symbolic execution to generate proof obligations, which are passed on to Z3 [7]. Note that because VerCors uses Viper as an intermediate representation, all that we can do to improve the performance is to change the Viper encodings; we never generate Z3 specifications directly.

We first describe how we identified the main performance bottlenecks in VerCors, and then we describe our solutions to these problems. In particular, we identify that (1) the encoding of arrays into Viper, and (2) the use of quantifier expressions in specifications without triggers to inform Viper on how to guide its underlying prover[3], were two of the main performance bottlenecks. To address these issues we developed a new array encoding, which also was more suitable to reason about multi-dimensional arrays, and took advantage of several recently developed features in Viper, such as the support for nested quantifiers over access permissions [16]. Furthermore, we implemented a technique to automatically generate triggers during the transformation from VerCors to Viper.

Of course, the contributions of this paper are mainly VerCors-specific, as we focused on improving the performance of VerCors. However, we believe that also some general lessons can be extracted from our experiences. In particular, we have developed a simple way to count subexpressions that help identify performance bottlenecks, which would be reusable for other program verification tools as well. Furthermore, most verifiers deal with arrays and quantifiers in some way, so we believe that also the lessons learned will be useful for other tool developers.

The remainder of this paper is organised as follows. Section 2 gives a high-level overview of the VerCors architecture. Section 3 discusses how we identified performance bottlenecks. Section 4 presents the old and new array encoding and evaluates their performance. Section 5 discusses how we automatically generate triggers to guide the verification of quantified expressions, which is then also evaluated experimentally. Finally, Sections 6 and 7 discuss related work and conclude. More details about the work described in this paper, as well as more extensive performance comparisons, are available in the Master's thesis of the first author [15].

2 Background: VerCors Implementation

VerCors currently verifies programs written in multiple concurrent programming paradigms, such as heterogeneous concurrency (C and Java), and homogeneous concurrency (OpenCL) [5]. In addition, it can also reason about compiler directives as used in deterministic parallelism (OpenMP).

[3] Triggers are a widely-used technique to give hints about the instantiation of quantifiers to an automatic prover.

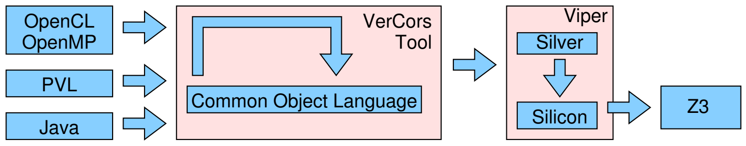

Fig. 1. Architecture of the VerCors tool set

VerCors uses a specification language based on the Java Modeling Language (JML) [6] extended with separation logic-specific notations [17]. Programs are annotated with pre- and postconditions that specify its intended behaviour, and the tool checks whether the implementation respects the annotations. For detailed information about the VerCors specification language we refer to the VerCors website at `http://www.utwente.nl/vercors`.

VerCors is built as a transforming compiler for specified code: it transforms programs in one of the VerCors input languages into an input language for a back-end verifier (currently Viper [16]). Figure 1 outlines the overall VerCors architecture. This architecture is set up to make it easier to create new (language) front-ends, to experiment with other transformations, and to use new back-end verifiers.

VerCors has front-end parsers for (concurrent) Java and C, for programs with OpenMP compiler directives and for OpenCL kernels. These parsers are extended with parsers for the specification language as used in VerCors. Further, there is a parser for the PVL language: the internal VerCors prototyping language. Each parser parses the annotated source program and produces an abstract syntax tree (AST) in the Common Object Language (COL). The AST of the COL language consists of nodes that represent the various concurrency abstractions that are supported by VerCors.

Transformations on the AST rewrite it to a tree that consists of nodes that can easily be mapped to the language that is used in the chosen back-end verifier. Two techniques are used to transform the AST. First of all, visitor patterns are used to replace a high level abstraction node by a semantically equivalent structure of simpler nodes. These transformations replace language constructs by their definitions, such as the flattening of expressions into a list of commands according their execution order. Second, a collection of standard rewrite rules is provided for rewriting into equivalent expressions, for example to make verification easier. To ensure that these transformations preserve the semantics, these transformations are taken as input from a separate file, making it easier to review and validate them, a process currently done by hand. It also makes it easy to extend the set of transformations. Every transformation is defined in its own *compiler pass*. The passes are referenced by a name, which makes it easier to reuse transformations, and to develop transformations for different back-end verifiers with other (syntactical) requirements.

The back-end verifier is then used to verify the transformed program. Viper programs, written in the input language Silver, are symbolically executed using

Silicon. The resulting first-order proof obligations are then given to Z3. To map possible errors back to the original source program, VerCors keeps track of where each node in the AST originated from. The only back-end that is currently supported is Viper [16], which is a intermediate verification language infrastructure that supports access permissions. This makes it highly suitable as a back-end for tools that use (permission-based) separation logic as specification language. Viper has two reasoning modes: one using symbolic execution (named Silicon) and one using verification condition generation via an encoding into Boogie (named Carbon). Both back-ends in Viper make use of the SMT (satisfiability modulo theories) solver Z3 [7] to discharge proof obligations.

3 Analysis of Performance Bottlenecks

In order to improve the performance of the VerCors verifier, we must first understand where the tool lacks performance. Therefore we talked with users and developers, both of VerCors and Viper, and we collected programs that were slow to verify. In particular, we interviewed all the PhD students in Twente that use and develop VerCors, and asked them for particularly slow programs. Their answers also included programs obtained from their experiences with supervising individual students on verification projects. In addition, we also interviewed one of the main Viper developers, who explained about the performance bottlenecks they had observed in Viper. Even though this group is limited in size, we believe that their answers are representative for the problems, in particular because they also included experiences from student projects.

We then compared the runtimes of the programs to the elements of VerCors that these programs use. We count the use of VerCors features by counting subexpressions in the AST, through a modification of the VerCors tool. As an example, consider the following expression, which specifies that all elements in the array named `input` are zero:

```
(\forall int i;
    0 <= i && i < input.length; input[i] == 0);
```

We count the following subexpressions (13 in total):

- expr.BindingExpression:Forall (1)
- expr.constant.ConstantExpression (2)
- expr.NameExpression (4)
- expr.OperatorExpression:LT (1)
- expr.OperatorExpression:LTE (1)
- expr.OperatorExpression:And (1)
- expr.OperatorExpression:Length (1)
- expr.OperatorExpression:Subscript (1)
- expr.OperatorExpression:EQ (1)

We used this to inspect many of the programs in our standard example set. This way of measuring performance allows us to see if certain VerCors features indeed correlate with slow runtimes. VerCors users have stated that arrays, sets, bags (both datatypes for specification), barriers (a synchronisation primitive), and quantifiers can yield slow performance. The Viper developers have also pointed to quantifiers as a likely bottleneck for performance.

Our use of programs for confirming performance bottlenecks is highly biased, for the following reasons:

- The sample size for many constructions is very small. Many VerCors constructions are tested by one or several unit tests in the example directory, and these unit tests are often their only use. Verification of such small instances is typically trivial, and hence the verification time for those constructions might not be representative.
- The programs we collected are biased towards constructions that have good performance. VerCors users are sometimes forced to make verification times faster for pragmatic reasons, and only the versions that are among the fastest will end up in our example set.
- The VerCors users who were asked to identify bottlenecks also created many of the programs. Thus, when we use those programs to confirm potential bottlenecks, we use a biased sample.

We nevertheless compared the performance of our programs to the VerCors features in it. Our experiments suggest that arrays and forall quantifiers are indeed bad for performance. For the other constructs, e.g. sets, bags, barriers, and the existential quantifier, there were not enough programs to confirm or deny this.

In the remainder of this paper we show the effects on overall performance when changing how arrays and forall quantifiers are treated. As certain programs indeed showed substantial speedups due to this change, we will conclude that at least in those programs, arrays and forall quantifiers have been bottlenecks.

4 Array Encoding

As discussed above, our verification time analysis indicated that the treatment of arrays was one of the potential performance bottlenecks. Therefore we changed the encoding of arrays into Viper. This section first briefly describes the old encoding, and then explains how a better encoding helps to improve the verification times.

4.1 Old Array Encoding

In the original array encoding, the parsing phase transforms all Java, C and PVL arrays into COL arrays. COL arrays are then transformed into Viper, using the Viper built-in notion of sequences. First of all, every sequence is wrapped by an Option type. This enables reasoning about both initialized and uninitialized arrays: OptionNone models an uninitialized array (e.g. a C array pointing to NULL or a Java or PVL array being null), while OptionSome contains the initialized array. In order to reason about permissions to elements in the array, the

$$\begin{tikzcd}
\boxed{\texttt{Int[]}} \arrow[r, "\text{parse}"] & \boxed{\texttt{Array<Integer>}} \arrow[r, "\text{rewrite}"] & \boxed{\texttt{Option<Seq<Cell<Integer>>>}}
\end{tikzcd}$$

Fig. 2. Steps of the old array encoding

```
input != VCTNone() && (|getVCTOption1(input)| == N
   && (forall i: Int, j: Int :: true
        && (0 <= i && i < |getVCTOption1(input)|
            && 0 <= j && j < |getVCTOption1(input)|
            && getVCTOption1(input)[i] ==
                getVCTOption1(input)[j])
        ==> i == j))
```

Listing 1. Viper encoding of `\array(input, N)`.

inner type of the array is wrapped in a Cell type. Eventually this Cell type is transformed into a reference to a field in Viper, which allows us to reason about permissions on this field. These steps are summarised in Figure 2.

Multi-dimensional arrays (arrays of arrays) are flattened into a one-dimensional array. In addition, a function is generated, which is used to calculate the index of the element in the flattened array that corresponds to the element of the multi-dimensional array. This flattening was necessary because previous versions of Viper did not allow nested quantifications, or quantification with multiple variables, for permissions. However, it also made the verification more complex, because the function to compute the index in the flattened array is non-linear, and reasoning about non-linear functions is undecidable and not well-supported by Z3.

As mentioned above, verification of arrays was identified as a performance bottleneck. Our investigations and discussions with the Viper developers indicated that this was in particular due to the predicate that we used to specify *injectivity* of the array encoding. By construction, arrays in Java and C are injective, thus every element in the array is different from all other elements in the array (every slot is a distinct block of memory). If we encode arrays as a sequence of references, this in not necessarily the case, since a sequence can look like e.g. `xs = 0X4, 0X4, 0X8, ...`, where the first two elements are references to the same block of memory. Therefore VerCors contains predicates to specify that an array (or matrix) is valid (keywords: `\array(<name>, <dim>)` and `\matrix(<name>, <dim1>, <dim2>)`), where valid means that the array or matrix is not null, has the specified dimensions and all slots are different. With the old encoding, the predicate `\array(input, N)` was encoded in Viper as shown in Listing 1 (with a similar encoding for `\matrix(<name>, <dim1>, <dim2>)`).

The problem is that Z3 has difficulties reasoning about the quantified expressions. Typically, provers allow the user to provide triggers to guide the proving process [8], in particular providing hints about how to instantiate quantifiers. In Viper, if no triggers are specified, it will try to infer suitable triggers (VerCors does not support triggers in the input language, since additional transformations on the AST may render the triggers unsuitable). For this function, in the body of the quantified expression there are two array accesses: for the element at index i and for the element at index j. Therefore Viper infers two triggers for this expression (matching on the access to element i and on the access of element j). As a result, during the proving process for every element of the array two instances of the expression are created, thereby creating a possible quadratic blowup of relations that the prover has to maintain, and this causes verification to become slow.

4.2 New Array Encoding

To improve support for the verification of multi-dimensional arrays, and to avoid the performance bottleneck of reasoning about array injectivity, we develop a new encoding of arrays.

First of all, as in the more recent versions of Viper it is possible to use universal quantifiers for expressions with multiple quantified variables, it is no longer required to flatten multi-dimensional arrays. Thus, also the non-linear function to calculate the index into the flattened array is no longer required, if we encode multi-dimensional arrays directly as multi-dimensional sequences.

To model the injectivity of arrays, we created the domain encoding in Listing 2. In this encoding every element of an array is modeled by the `loc` function, which combines a `VCTArray` object with an index. The domain is parameterized with the `CT` type as the type for the elements in the array. In the cases where we want to reason about permissions on elements in the array, this can be a reference to a field, but it can also be any other type as defined in the resulting Viper program. The functions `first` and `second` are used to retrieve the `VCTArray` object and the index from an array element. These functions are only used internally by the axiom `all_diff`. In the `all_diff` axiom, the relation between on one side the `VCTArray` plus an index, and on the other side the corresponding element in the array is made explicit for all elements in the array, thereby encoding injectivity of the array. Note that the axiom `all_diff` has only one trigger (`loc(a, i)`), rather than the two triggers that were inferred in the old array encoding to specify injectivity. Further there is a function `alen`, which models the length of the array.

As multi-dimensional arrays are no longer flattened, we now also have to consider the encoding of the inner arrays within a multi-dimensional array. For simplicity, we will explain this in terms of a 2-dimensional matrix, using the term row to refer to an inner array within the matrix[4]. Just as in the old encoding we wrap the outer array (matrix) in an Option type, to be able to reason about Java arrays being null or C arrays pointing to NULL. For the inner arrays (rows) there is a choice to be made. For Java the rows could also be null. In C there are variants where rows can be NULL (the matrix is a list of pointers to rows), and variants where rows can not be NULL (the matrix is flattened to a single

[4] The implemented encoding works for arbitrary multi-dimensional arrays.

```
domain VCTArray[CT] {
  function loc(a: VCTArray[CT], i: Int): CT
  function alen(a: VCTArray[CT]): Int
  function first(r: CT): VCTArray[CT]
  function second(r: CT): Int

  axiom all_diff {
    forall a: VCTArray[CT], i: Int :: { loc(a,i) }
      first(loc(a,i)) == a && second(loc(a,i)) == i
  }

  axiom len_nonneg {
    forall a: VCTArray[CT] :: { alen(a) }
      alen(a) >= 0
  }
}
```

Listing 2. Viper domain for new array encoding

$$\begin{tikzcd}
\boxed{\texttt{Int[][]}} \arrow[d, "\text{parse}"] \\
\boxed{\texttt{Array<Array<Integer>>}} \arrow[d, "\text{rewrite}"] \\
\boxed{\texttt{Option<VCTArray<Option<VCTArray<Cell<Integer>>>>>}}
\end{tikzcd}$$

Fig. 3. Steps of the new encoding for a 2-dimensional matrix

array during compilation). In order to make a simple and uniform translation into Viper, the choice was made to always wrap rows in Option types in the COL language. This allows to capture both the different kinds of C arrays, as well as Java arrays within the same language construct. It is future work to investigate if we can fine-tune this encoding: with the current approach we loose some information about the original array in the source language.

As we do not see the necessity to reason about permissions or values of entire rows in a matrix, the rows are not wrapped in a Cell type. However, the inner types of the rows are being wrapped in a Cell type, which makes it possible to reason about permissions to the elements of the matrix. Figure 3 summarises the transformation steps for a 2-dimensional matrix.

With this new encoding, it is no longer required to explicitly specify injectivity for valid arrays and matrices, as this is implicitly encoded by the `all_diff` axiom in the `VCTArray` domain (Listing 2). Thus the Viper function that encodes the valid array property now only states that the reference is non-null and that the array has the specified length. For a valid matrix a non-null condition is generated for the outer array, together with the condition that the matrix has the specified number of rows. Additionally, for every row in the matrix it generates a non-null constraint and a condition that the row has the specified number of cells (columns). Notice that if we would have wrapped the rows in Cell types as well, then Viper could no longer deduce that all rows are different, since each row would now be encoded as a reference to a field with an `Option <VCTArray<Cell<Integer>>>` type, so that two elements of the outer array of the matrix could point to the same referenced field.

4.3 Experiments

In order to measure the effect on performance of verification using the new array encoding, a set of programs has been selected from the VerCors example repository. This set of programs contain all the different concurrency abstractions and data types that are supported by VerCors and for which programs are available. Thereby it is possible to see if the new array encoding might also have unforeseen effects on the verification of other program or specification constructs. In the experiment each example is verified 5 times, and the mean time is used to compare results for the different versions of the tool.

4.4 Results

Because of updates to VerCors and updates to the Viper back-end, several versions have been compared. We indicate the Viper version by a number, and when we have used the new array encoding this is indicated by a suffix `-a`. This gives the following versions:

– `Vct1`: Version of the tool for which all the programs verify. Used as a base-line for the experiment.
– `Vct2`: Version of the tool with an updated version of the Viper back-end. Required to make use of nested quantification for multi-dimensional arrays.
– `Vct2-a`: Version `Vct2` with the new array encoding.
– `Vct3`: Updated version of VerCors with a later update to the Viper back-end, with the old array encoding.
– `Vct3-a`: Version `Vct3` with the new array encoding.

Table 1 shows the results of our experiment to compare the verification times using the old array encoding and using the new array encoding. The new array encoding is based on version `Vct2` of the tool. However, due to a non-linear calculation that was needed in the old array encoding to calculate the new index in a flattened array for a multi-dimensional array, it was no longer possible to verify some programs with multi-dimensional arrays with this version of the tool. We also show results for an earlier version of the tool, version `Vct1`, in which all benchmarks verify.

Using the new array encoding we can see that we are now able to verify the `case-studies/prefixsum-drf.pvl` and the `carp/histogram-submatrix.c` again. Further we see that the `case-studies/prefixsum-drf.pvl` example is almost four times faster to verify, from version `Vct1` to version `Vct2-a`. This

File	`Vct1`	`Vct2`	`Vct2-a`
`case-studies/prefixsum-drf.pvl`	193286	-	50141
`carp/histogram-submatrix.c`	19579	-	17664
`verifythis2018/challenge2.pvl`	21709	-	-
`carp/summation-kernel-1.pvl`	24320	18896	16817
`manual/option.pvl`	6798	8876	8364
`waitnotify/Queue.pvl`	5441	7518	7116
`type-casts/TypeExample1.java`	5261	7151	6776
`basic/CollectionTest.pvl`	5047	7570	7308
`witnesses/TreeWandSilver.java`	33340	32210	31800
`layers/LFQHist.java`	13597	15113	15111
`arrays/DutchNationalFlag.pvl`	10799	13019	13024
`futures/TestFuture.pvl`	6824	8401	8656
`floats/TestFloat.java`	15827	15542	16393
`openmp/add-spec-simd.c`	14440	14127	16136
`openmp/addvec2.pvl`	12822	13312	16037
`floats/TestHist.java`	21423	17862	22361

Table 1. Comparing total verification times (in ms) of `Vct1`, `Vct2` and `Vct2-a`. Sorted by relative speedup from `Vct2` to `Vct2-a`.

is the example that was pointed out by users to be slow to verify, and in which the experts identified the array encoding as being a possible performance bottleneck. The `verifythis2018/challenge2.pvl` example no longer verifies in version `Vct2` and `Vct2-a` of the tool. Upon closer inspection, this is not related to the array encoding, but to a bug in the Viper backend in verifying a loop-invariant: the same program would sometimes verify and sometimes fail. We think that optimizations for sequences in Viper may explain the degradation in performance for the `floats/TestHist.java` and `openmp/addvec2.pvl` programs, since in the new array encoding we no longer make use of sequences. Future work may be to investigate if we can apply possible similar optimizations as for sequences to our array encoding.

In Table 2 we compare results from version `Vct2` to version `Vct3` and `Vct3-a`, which are later versions of VerCors with an update to the Viper back-end. Overall we see a performance decrease after the Viper update. With the new version of Viper used in `Vct3` and `Vct3-a`, we can no longer unambiguously claim that the new array encoding improved the performance. While there are two benchmarks that only work when the new array encoding is used, there are also some benchmarks that clearly become slower under the new encoding. Overall, this Viper update has degraded performance.

Furthermore, the `floats/TestFloat.java` example no longer verifies in version `Vct3` and `Vct3-a`. Verification of this example does not terminate due to a matching loop in a quantified expression in the specification. Matchings for quantifier instantiations in general influence verification performance. We look at this in detail in Section 5.

File	`Vct2`	`Vct3`	`Vct3-a`
`case-studies/prefixsum-drf.pvl`	-	-	98715
`carp/histogram-submatrix.c`	-	-	18095
`floats/TestFloat.java`	15542	-	-
`carp/summation-kernel-1.pvl`	18896	19694	18934
`manual/option.pvl`	8876	10275	10103
`basic/CollectionTest.pvl`	7570	8483	8391
`waitnotify/Queue.pvl`	7518	8283	8197
`type-casts/TypeExample1.java`	7151	8022	8068
`witnesses/TreeWandSilver.java`	32210	29072	29334
`arrays/DutchNationalFlag.pvl`	13019	13887	14036
`futures/TestFuture.pvl`	8401	9568	9741
`layers/LFQHist.java`	15113	16130	16481
`openmp/add-spec-simd.c`	14127	15514	18842
`openmp/addvec2.pvl`	13312	14673	18530
`floats/TestHist.java`	17862	20226	34278

Table 2. Comparing total verification times (in ms) of `Vct2`, `Vct3` and `Vct3-a`. Sorted by relative speedup from `Vct3` to `Vct3-a`.

5 Trigger Generation

As mentioned earlier, VerCors has no support for triggers in the input language. The main reason for this is that VerCors rewrites the AST, and could thereby invalidate triggers. For instance, to encode parallel blocks, VerCors generates new quantifiers, and these quantified variables are not covered by the triggers that were in the input program. Thus VerCors relies mainly on Vipers capability of inferring triggers. However, to improve the situation, we have to investigate if during the transformation we could generate some triggers explicitly, and if this would improve performance.

Quantifiers in Z3 are instantiated via a mechanism called triggers [8]. The idea is to associate every quantified expression with a trigger set. We focus on the case that a quantified expression like $\forall x.E(x)$ (and its associated trigger set) becomes a positive literal in Z3's procedure. When this happens, Z3 can introduce values for x, say $c_1, \ldots, c_n$, and add $E(c_1), \ldots, E(c_n)$ to the set of positive literals. The trigger set helps Z3 to determine which values for x to use. A trigger is an expression with the meaning: if this expression is unifiable with a ground term that occurs in any literal, use that ground term to instantiate x. A trigger set is a set of such expressions.

5.1 Rewriting Complex Index Expressions

In the general case, a quantified expression can have multiple variables:
$Q\, x_1, \ldots, x_N, E$. In order to generate triggers for Viper, which passes them down to Z3, we have to make sure that they adhere to the restrictions that Viper imposes on triggers (see [18] for more information on these restrictions):

```
class Subscripts {
  invariant (\forall int i; 0<=i && i<|s|/2; s[i] == s[2*i]);
  void fun(seq<int> s);
}
```

Listing 3. Function specification with complex index expression.

- All quantified variables occur in the trigger set.
- Each trigger contains a quantified variable.
- Each trigger contains a function symbol: a single quantified variable by itself is not a valid trigger.
- The trigger expression does not contain any arithmetical operator (like addition on integers).
- Triggers do not contain accessibility predicates, i.e. you are not allowed to use permission expressions in your triggers.

This can be challenging, because data structures that are often used in combination with quantified expressions are arrays or sequences. However, we cannot generate triggers of the form `input[i+1]`, since the index part of the expression uses the `+` operator, and this is not allowed by Viper. Thus, as a first step, we developed a rewriting procedure to rewrite index expressions in the allowed format.

To illustrate this rewriting procedure, consider Listing 3.

In this example the forall quantifier specifies that for all elements in the first half of the sequence, the element at the position twice as far from the start should have the same value. In this example there are two candidate trigger expressions: `s[i]` and `s[2*i]`. This is not a valid trigger set, since the second expression contains the arithmetic operator `*`. To eliminate the multiplication, we use rewriting to (1) introduce a fresh quantifier variable `u1`, (2) replace the `2*i` expression in the body of the expression with the new variable, and (3) add an equality to the range expression of the quantifier, stating that the new variable should be equal to `2*i`. This results in the following expression.

`(\forall int i,int u1; 0<=i && i<|s|/2 && u1==2*i; s[i]==s[u1])`

This gives us the candidate triggers `s[i]` and `s[u1]`, which adhere to the Viper restrictions on triggers.

One point of consideration is the effect of the complexity that we add to the quantifier expression by adding a quantified variable. The extra variable adds a dimension to the domain of the quantifier, which could make it even harder for the SMT solver to find the right instances to discharge a proof. We believe, however, that the SMT solver can easily discharge the added equality in the selection of the quantifier (in our example the equality `u1 == 2*i`), and that the positive effect of being able to generate an appropriate trigger will outweigh the costs of the added complexity. Another way in which we add complexity, is that by adding an extra trigger (in this case for `s[2*i]`), more clauses are added. This is typically a desired additional complexity, as it enables us to prove more properties (Z3 applies the clause in more situations), but it has a potential to backfire by causing time-outs. Our experiments show that this is the case, and that there are programs where rewriting helps the prover, whereas in other programs rewriting causes the verification to fail.

5.2 Generating Triggers

After eliminating complex index expressions, we can try to generate trigger sets for universal quantifiers at the end of the rewriting phase (as this ensures that the generated triggers will not be invalidated by other transformations). Both the range and the body of the quantifier expression are trigger candidates. From these subexpressions we identify all trigger candidates: expressions that mention at least one of the quantified variables, have some sort of structure (thus not the variable itself) and do not contain an accessibility predicate. We then generate the powerset of all trigger candidates, and from this select all sets that mention all quantified variables.

We chose to consider all possible valid combinations of trigger expressions, to make sure that we do not inadvertently block necessary quantifier instantiations. The intuition is that these triggers still contain more abstractions (e.g. domain encodings used by VerCors) than triggers that would be inferred by Viper. Thereby our triggers are more specific and cause less spurious quantifier instantiations in the SMT solver. It is future work to investigate if some minimalisations are possible. However, our experiences are that starting with the full powerset does not have a significant overhead.

5.3 Experiments

We used the same set of programs as in Section 4.3 to compare verification times using the rewriting and trigger generation techniques discussed in this section. Version `Vct3-a` is the version of VerCors in which all the aforementioned techniques are implemented. This version already makes use of the new array encoding as introduced in the previous section. Using a command line option, we can enable rewriting of complex indexes and the generation of triggers separately. For each configuration the programs have been verified 5 times, and the mean verification times are shown in the table.

5.4 Results

Table 3 shows the verification times for our experiments. Column `Vct3-a` shows the verification times without rewriting index expressions, and without trigger generation. In column `Vct3-a-t1` only trigger generation is enabled. In column `Vct3-a-t2` only rewriting complex index expressions is enabled and in the column `Vct3-a-t3` both the rewriting of complex index expressions and trigger generation are enabled.

File	`Vct3-a`	`Vct3-a-t1`	`Vct3-a-t2`	`Vct3-a-t3`
`floats/TestFloat.java`	-	32693	-	34219
`openmp/add-spec-simd.c`	18842	19329	-	-
`openmp/addvec2.pvl`	18530	19629	-	-
`floats/TestHist.java`	34278	24412	31877	23122
`case-studies/prefixsum-drf.pvl`	98715	96133	68190	69797
`arrays/DutchNationalFlag.pvl`	14036	14153	14673	13975
`manual/option.pvl`	10103	10266	11123	10072
`type-casts/TypeExample1.java`	8068	7993	8002	8081
`futures/TestFuture.pvl`	9741	9880	9708	9819
`layers/LFQHist.java`	16481	16457	16662	16657
`carp/histogram-submatrix.c`	18095	18380	18058	18324
`basic/CollectionTest.pvl`	8391	8289	8394	8500
`carp/summation-kernel-1.pvl`	18934	19517	21016	19222
`waitnotify/Queue.pvl`	8197	8190	8304	8351
`witnesses/TreeWandSilver.java`	29334	29502	30290	30001

Table 3. Comparing total verification times (in ms) for version `Vct3-a`, with and without triggers. Sorted by relative speedup from `Vct3-a` to `Vct3-a-t3`.

There are a couple of things we can see from the results. Without triggers `floats/TestFloat.java` does not verify. Verification for this program runs indefinitely due to a matching loop in the back-end SMT solver. By generating triggers for the quantified expressions in this program, the matching loop is broken and the tool is able to verify it. Verification of `floats/TestHist.java` is nearly 30 % faster if triggers are generated. For the other results in the `Vct3-a-t1` column we see minimal differences in verification times, because we can not generate valid trigger sets for the quantified expressions in these programs. If we enable the rewriting of complex indexes, we see that the `openmp/add-spec-simd.c` and `openmp/addvec2.pvl` programs fail to verify. This is caused by the added complexity introduced by adding quantified variables for complex index expressions. For `case-studies/prefixsum-drf.pvl` we see that only rewriting complex index expressions already has a positive effect on the verification time. Combining the two techniques show positive results for 3 of the 15 programs, but cause 2 programs to fail. Thus the use of triggers show a positive effect on performance. However, transformations that are needed to enable trigger generation can have negative side-effects. Therefore future work could focus on how to rewrite quantified expressions in a way that enables trigger generation, without adding too much complexity.

6 Related Work

The work in this paper is focused very specifically on the performance of program verification with VerCors. However, if we look more broadly, we can see various studies that investigate and compare the performance of different verification techniques, as discussed below. To the best of our knowledge, there are no other reports that analyse specifically which parts of an annotated program have most impact on the performance of the program verifier.

Kassios et al. compared the performance of verification using symbolic execution with the performance of verification condition generation-based verification [12]. Their work compares different technique to obtain the necessary first-order proof obligations, whereas we compare the performance effect of different Viper encodings for the same source program.

Leino et al. identify matching loops in quantified expressions as a significant contributor to instabilities in performance and user experience in program verification [14]. They propose to move trigger logic away from the SMT solver and into the high level verifier. In particular, they present three techniques for trigger selection that they then implemented in the Dafny verifier: quantifier splitting, trigger sharing and matching loop detection. In our work, we have also investigated ways to generate suitable triggers at a higher level. Our results also show a performance benefit, but still requires more investigation to understand all possible consequences.

7 Conclusions

In this paper we have discussed two techniques to improve the run-time performance of the VerCors program verifier: changing the encoding of arrays, and changing the way triggers are generated. This gives a good speedup on some programs from our standard example set, but on many programs, the run-time hardly changes. A summary of the measurements presented in this paper is given in Figure 4. However, our benchmarks come from a set of programs that were selected to work well with the version of VerCors that does not have the improvements described in this paper. We therefore believe that the changes, which do help in some cases, will have a large positive impact on the practical use of VerCors. In particular, as our new encoding is simpler because of the implicit injectivity, we expect that in the future we might have to add less auxiliary lemmas about the bounds on array index expressions. Moreover, we should also stress that the new array encoding also enabled an easier encoding of the various kinds of (multi-dimensional) arrays one finds in Java and C. As future work, we plan to further fine-tune our improvements to further improve the performance of VerCors.

Acknowledgements

The work in this paper has been supported by the NWO VICI project Mercedes (project nr. 639.023.710).

References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. *Deductive Software Verification – The KeY Book*,

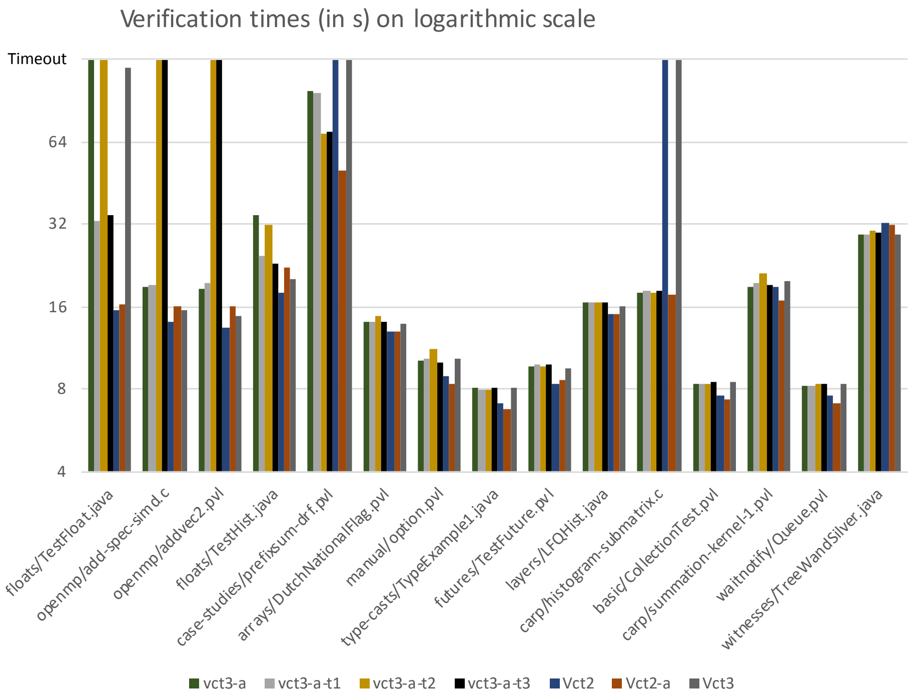

Fig. 4. Summary of Tables 1, 2, and 3

volume 10001 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016.

2. A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In *Programming Languages meets Program Verification (PLPV 2012)*, pages 71–82, 2012.
3. Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of concurrent systems with VerCors. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 172–216. Springer, 2014.
4. S.C.C. Blom and M. Huisman. The VerCors tool for Verification of Concurrent Programs. In *FM*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
5. Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve Schneider, editors, *integrated Formal Methods (iFM)*, pages 102–110. Springer, 2017.
6. Lilian Burdy, Yoonsik Cheon, David R Cok, Michael D Ernst, Joseph R Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
7. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

8. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
9. Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 345–373. Springer, 2019.
10. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, 2008.
11. Bart Jacobs, Jan Smans, and Frank Piessens. Solving the VerifyThis 2012 challenges with VeriFast. *International Journal on Software Tools for Technology Transfer*, 17(6):659–676, Nov 2015.
12. I. T. Kassios, P. Müller, and M. Schwerhoff. Comparing verification condition generation with symbolic execution: an experience report. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software Theories Tools Experiments (VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 196–208. Springer-Verlag, 2012.
13. K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.
14. K Rustan M Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *International Conference on Computer Aided Verification*, pages 361–381. Springer, 2016.
15. Henk Mulder. Performance of program verification with VerCors. Master's thesis, University of Twente, 2019.
16. Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.
17. John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.
18. Viper tutorial. Last visited: 09-04-2020.