University of Twente

Final project

Performance of program verification with VerCors

Henk Mulder

supervised by
Prof.dr. M. Huisman
Dr.ir. S.J.C. Joosten

July 2, 2019

Abstract

Program verification is only as useful as its ability to produce results in a timely manner. In this research we investigate what performance bottlenecks are in the VerCors verification tool for concurrent programs. The aim is to identify the cause of a performance bottleneck, in order to optimize the tool.

We introduce a technique to identify what properties of a program are more difficult to verify. Using those results, we present solutions to two performance bottlenecks that were identified: 1. An alternative encoding of arrays is implemented in the tool which allows the tool to reason up to 4 times faster about programs that make use of arrays. 2. Our research in generating triggers for quantified expressions show that speedups up to 30% are possible. Though further research is required to investigate if this solution can be generalized and optimized further.

Contents

Chapter 1

Introduction

Computer software takes an increasingly important role in our life. Think of the software that is used to (help to) fly planes, that is used to control our financial systems, but also that is starting to drive our cars. Therefore it is more important than ever to make sure that the software works as intended. However, the tasks that we can "give" to computers are getting more and more complex. That is made possible by the leaps in processing power that are made in the semi conductor industries that produce our computer chips. With multi-core processing units it is possible for software to perform tasks in parallel, where multiple processes are working on multiple (different) tasks. However, one process might run faster than others, e.g., because of scheduling by an operating system or other external factors. If multiple processes then also have to work on data that is shared between the processes it becomes inherently harder to show that the tasks are performed as intended, since all possible interleavings of the processes have to be considered.

Fortunately, it is possible to use the same computing power to analyze and mathematically verify the software. This is done by so-called deductive software verification. From the program and specification, we derive verification conditions. If the verification conditions hold, the program respects its specifications. The verification conditions can be discharged (or checked) using automated theorem provers.

A specification language is used to specify what the "intended workings" of the program are. This is done by adding annotations to the program that specify what conditions should hold for the state of the program at specific points. These annotations can be pre conditions (conditions that should hold before a method call), post conditions (conditions that will hold after the method call) or invariants (conditions that hold before, during and after a method call). Additional assertions and loop invariants can be used to specify (required) properties in intermediate points of the program.

Verification is based on Hoare logic [10]. In this logic a Hoare triple $\{P\}S\{Q\}$ binds together a precondition P, a program statement S and a postcondition Q. If the Hoare triple is valid this means that executing S in any state that satisfies

P, on completeness the result state will satisfy Q. Applying this technique by enclosing program methods as statements with pre- and post conditions then gives the means to reason modularly about correctness of programs.

In order to use program verification to improve the quality of the software that is being developed, the verification techniques need to be integrated into a development workflow. That means that first of all the technique needs to be integrated in the (higher level) programming language in which the program is developed. Next, the specification language needs to be expressive enough for the developer to express the desired properties of the software with reasonable effort. And finally it should produce results that are of use to the developer. In the VerCors tool this is done by supporting Java and C (to work with OpenCL and OpenMP). A specification language based on JML is used to specify the desired properties. See (the left half of) figure 1.1. Verification results are translated to descriptive messages and when applicable mapped to the relevant parts of the source code.

One of the key parts of making a verification tool usable is the speed with which it manages to produce results. In this regard the VerCors tool seems to perform somewhat unpredictably. For instance, writing the same property over a (possibly unbounded) number of heap locations using recursive predicates instead of using quantifiers (or visa versa) can influence the verification time a lot. In extreme cases, we have seen that a program that could be verified in approximately 10 seconds with one technique could take more than 5 minutes to verify using the other technique. With this research we will look into possible causes for this unpredictable behavior, how we can identify root causes and how to possibly reduce the problems.

The project focuses on two main questions:

- What are the performance bottlenecks in VerCors?
- How can we mitigate found performance bottlenecks, using the techniques and tools that are available in VerCors?

To answer these questions we consulted users of the VerCors tool and experts on the underlying verification tools. To help to answer the first question we introduce a framework to identify which aspects of a program are slow to verify with VerCors. With this framework we can identify specific syntactical structures in input programs, and monitor a normalized value of the time it takes to verify programs that contain this structure. By comparing this value against the normalized values for other structures we can determine if the structure is more likely to cause a bottleneck in verification. In answer to the second question we look at two bottle-necks that were identified: The encoding that is being used to reason about arrays and the way VerCors treats universal quantifiers. For the former we propose a new array encoding that allows us to reason up to four times faster about arrays. For the latter we investigate if triggers can help to boost the performance when universal quantification is used in specifications. We show that triggers can have a significant positive effect on performance.

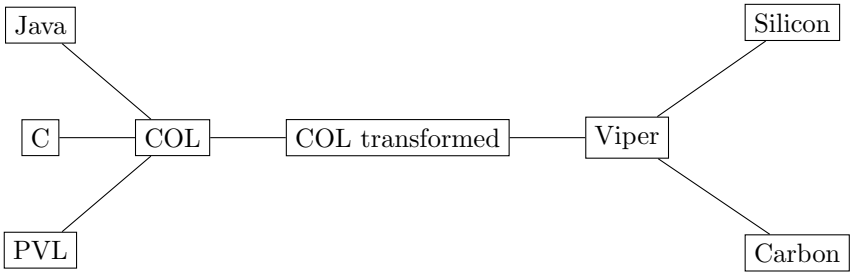

Figure 1.1: General VerCors architecture

However, we also see that transformations that are needed to be able to generate triggers can cause a negative effect. Therefore more research needs to be done to investigate if we can prepare for, and generate triggers only when it has a positive effect.

We continue with some background information on the VerCors tool in chapter 2. Next we discus what performance is in the context of program verification with VerCors, how we measure performance and how we have identified performance bottlenecks in Chapter 3. In Chapter 4 and 5 we discuss two performance bottlenecks that were identified; namely the encoding as used for arrays and the use of quantifiers without triggers. In Chapter 6 and Chapter 7 we look at related work and conclude with a summary of the results and we discuss future work.

Chapter 2

Background

The VerCors tool [1] is a tool to verify concurrent and parallel programs. It aims to make verification usable for developers that are not necessarily formal method experts. VerCors uses a specification language based on the Java Modeling Language (JML) [6] extended with notations for separation logic [15]. The tool currently verifies multiple concurrency paradigms, including heterogeneous concurrency (C and Java), compiler directives as used in deterministic parallelism (OpenMP) and GPU kernels with barriers and atomics (OpenCL) [4].

In this chapter we look at the theories as used in VerCors and how these are implemented/ used in VerCors.

2.1 Separation logic

As mentioned before, VerCors uses a specification language based on JML extended with annotations for separation logic. Separation logic [15], is an extension of Hoare logic to reason about low-level imperative programs that use shared mutable data structures. In separation logic there is an explicit distinction between the heap and the store. This means that you can express that you have a pointer to a location on the heap. The separating conjunction ($*$) is used to combine formulas. A formula $\phi_1 * \phi_2$ is valid for heap H if we can split the heap into two distinct heaps h_1 and h_2 in which ϕ_1 holds for heap h_1 and ϕ_2 holds for heap h_2. This makes that separation logic is also suitable to reason about multi-threaded programs. If threads operate on different parts of the heap, they can be verified in isolation. However in classical separation logic two threads are not permitted to read the same data. Something that is allowed, and often necessary in multi-threaded programs. Therefore in VerCors the separation logic is extended with access permissions.

2.2 Permission-based logic

To reason about which thread can read and write to certain locations on the heap, VerCors makes use of permission-based logic [5][9][1]. In permission-based separation logic every location on the heap is associated with a permission. Permissions to a heap location can be divided and combined, but can not be duplicated. To write to a heap location, a thread needs 100% of the permissions to that location. With a partial permission it is only possible to read a location. This ensures that if a thread has a read permission for a location, the value on that location will not change, since no other thread can have 100% of the permissions. On the other hand, if a thread has write permission to a location it will be the only thread that has permission to that location. Thus if all locations on the heap are protected with permissions then the program is data race free and reasoning can be done in a thread-modular way.

Listing 2.1: Example specification in C.

```
1 //@ requires Perm(x, read) ** Perm(y, read);
2 //@ ensures Perm(x, read) ** Perm(y, read);
3 //@ ensures \result == *x + *y;
4 int plus(int* x, int* y) {
5   return *x + *y;
6 }
```

To illustrate the concepts see Listing 2.1. In this code fragment, we have a method to add two integers that are stored on the heap (pointers x and y). In line 1, we specify that we need permission to read location x and location y. The permissions are separated by a double star, which is the symbol used by VerCors to denote the separating conjunction. The double star is used to distinguish it from the multiplication operator. In line 3, we specify a functional property, namely that if the method terminates the result is the sum of both integer values. In order to prove that the values at x and y are not changed by other threads during the execution of the method, we must prove that we have not lost the permissions on the heap locations. That is specified in line 2. The requires and ensures clauses specify the pre- and postconditions of the method, thereby making it possible to reason modularly about the code fragment just as it is done with Hoare triples. With these ingredients, VerCors can deduce that the implementation adheres to its specification. Of course, this is quite trivial for this example, but with this specification this piece of code can now be used in reasoning about other (more complex) programs that make use of this code. Note that VerCors also allows us to write the pre- and postcondition for the permissions in line 1 and 2 in one statement, using the context keyword. However, for illustrative purposes we have specified them separately.

2.3 VerCors implementation

The VerCors tool is built as a compiler for specified code. Currently VerCors has front-end parsers for (concurrent) Java and C, for programs with OpenMP compiler directives and 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; a program/ prototype verification language.

For every front-end programming language, a 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 the syntax tree to a semantically equivalent 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: With visitors parts of the tree can be changed, by replacing a high level abstraction node by a semantic equivalent structure of simpler nodes. Using a rewrite system, the AST can also be refined according to rewrite rules. The rewrite system tries to match the left-hand side of a rule, and if it matches it will be replaced by the right-hand side of the rule. This continues until no more matches are found.

Every transformation is defined in its own "compiler pass". The passes are referenced by a name, thereby making it easy to reuse transformations, and use them in new lists of transformations for back-ends with other (syntactical) requirements.

In the back-end, verification is done on the transformed program. To map possible errors back to the original source program, VerCors keeps track of where each node in the AST originated from. A simple overview can be seen in Figure 1.1. This architecture aims to make it easier to create new (language) front-ends, experiment with other transformations or to use other verification back-ends.

2.3.1 Viper back-end

Initially VerCors translated the given verification problems to Chalice [12] and Boogie [2]. Over time, the functionality of these back-ends were subsumed by the Viper tool [14].

Viper is a verification infrastructure with strong support for permission-based logics such as separation logic. It has support for two back-ends: 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.

The Viper intermediate language is a simple sequential language that includes a flexible permission model. This allows us to express the higher level concurrency abstractions as a Viper program, and use the Viper program for

both the symbolic execution back-end and the verification constraint generator back-end.

Versions of the tool

For the work in this thesis three versions of the VerCors tool play an important role. Work on this thesis started when version `Vct1` was current (Jun 2018). This version serves most as a reference for measurements and comparison to other versions. In version `Vct2` a (Sep 2018) start was made to address the first performance bottleneck; the encoding of arrays. In this version an update to the Viper back-end broke some of the functionality in VerCors. The work described in this thesis not only mitigates some of the performance bottlenecks that were present in version `Vct1`, it also fixes most of the problems that were present in version `Vct2`. And version `Vct3` is a more recent version of VerCors (Jan 2019), that is used to illustrate how the work for this thesis contributes to the tool. References to the actual sources of these versions can be found in Appendix A.

Chapter 3

Analysis of performance bottlenecks

In order to optimize performance of the VerCors tool, we must first establish where the tool lacks performance. The most obvious way to identify performance bottlenecks is to talk to users of the tool. In most cases the cause of a bottleneck is not immediately clear. To understand this we have to look at the process of writing the program specification. The user has an idea about the property of the program that he or she wants to verify. However, often the user has a choice in how to specify this property. For instance, let's assume the user wants to specify that all elements in an integer array have the value 0. The user could specify this property by using a universal quantifier of the form forall **int** i; 0<=i && i<a.length; a[i] == 0;, but this property could also be specified using a recursive function like **boolean** zero(**int**[] a, **int** i) = i<a.length? a[i]==0 && zero(a, i+1): **true**;. In different contexts, one or the other could be more suitable. When the specifications become more complicated, more choices are possible. Further specifications for one method might be needed to prove another property for a different method. Therefore a user might decide to change the specification in order to be able to proof another property. During this process, it is not always clear what effect the change to one specification has on proving the other specifications. Furthermore, a slow-to-verify specification for a program is undesirable, and is therefore either discarded or changed, leaving us with nothing to analyze. Therefore the most concrete evidence for a bottleneck in verification performance a user can give is an example with an annotated program that is slow to verify (despite the efforts to improve it).

Another useful source to help identify performance bottlenecks are Viper-experts. VerCors translates the input program to a Viper program, and the actual verification is done in the Viper tool. People with an expertise in verification with Viper could thus advise how to best encode certain constructs into Viper code.

In order to objectively reason about performance in relation to program ver-

ification with VerCors, we looked at more then just expert opinions. Therefore, in Section 3.1, we first define what performance is in this context. Next we explain how we measured performance in Section 3.2. Then, in Section 3.3, we look at the framework that was developed to identify what parts of a program are more or less slow to verify. In Section 3.4, we discuss the results and the performance bottlenecks we identified.

3.1 Defining performance

In order to identify bottlenecks in performance, we must first define performance in the context of program verification with VerCors. Since input languages, specification language and verification back-ends are already determined by what is currently available, performance is not about what is and what is not supported. What remains is the time it takes to verify a program. From a tools point of view it is simple. The tool receives an input (program) from the user, does its work and presents the results. From a user point of view it is more complicated. Verifying a program implicitly means that the program is a given entity, and is something we can not change. However using VerCors means that the user is in the process of verifying the program. This means that specifications and annotations change during this process. In this process, the tool can already steer or suggest the user to create the specifications and annotations in a certain way. This means that the tool can have somewhat influence on its input. In this research, we assume the input program to be a given entity. We want to evaluate performance on a mechanical level. We want the tool to perform optimally independent of the way how a specification is written, even if another specification by the user might give better performance.

3.2 Measuring results

In order to measure the effect of changes in the tool on task performance, scripts have been developed[1]. These scripts use the output of VerCors to determine the time it takes to verify a given program. By default, VerCors already reports the total verification time in milliseconds. By calling the tool with the `--progress` flag, VerCors will also report the time every intermediate pass in the tool takes. This is used to distinguish between the total verification time and the time that is spent in the back-end verification tool for the actual verification. Thereby we can tell where most of the verification time is spent, and where we thus need to focus our attention.

With the script to call the tool and process the outputs we compare verification times of different versions of the tool, using different encodings, on the same examples. Of course this makes it easy to do multiple runs and compute mean times and deviations.

[1]The VerCors scripts are located at `https://github.com/HenkMulder/vercors-scripts.git`

Representable examples

In order to determine how tool changes affect performance of the tool, a set of examples has been selected from the VerCors example repository. First of all, the selection was reduced by using the following criteria:

- Examples should work with the Silicon back-end.
- Each example consists of a single file.
- The examples have a Pass verdict.

Then we looked at the AST node types of these examples when they are first translated to the internal COL language. These AST node types represent the different concurrency abstractions that VerCors support, such as, parallel blocks and barriers. We also look at the types that are used in the examples. such as class types in Java or pointers in C or any other special type used for specifications, like sets or bags. In our final set of examples, we want to cover every type of COL AST node and all the types that are used in the examples. In this way we are certain that if a translation for one of the abstractions or types is changed, we can monitor the effect on all other types and abstractions as well.

First we looked at the examples that took more than 10 seconds to verify. This is because these examples are more likely to represent realistic verification problems, rather than being a test for one or another feature. Then we collected all node types, determined which type was most rare and added the example that has most of these nodes to our representable set. For the nodes and types that are not covered by the larger examples, we used the same procedure on the set of smaller examples until all node types and data types are covered by the set of representable examples. The resulting set is listed in Appendix B and has 16 examples in both Java, C and PVL.

Note that examples were selected from the VerCors repository at the time that version `Vct1` was current. In later versions of VerCors, a newer version of Viper is used that caused some examples to break. This concerns examples that make use of arrays and sequences. Some of these examples had to be updated, where others could not be fixed. The updates were merged into the repository at a later date. Therefore, all measurements are done on the representable example set that is current at the commit as described in Appendix B. All the examples in this version are backward-compatible with version `Vct1` of VerCors. The examples that could not be fixed for version `Vct2` and `Vct3` are there to illustrate the results of the work that is done for this thesis.

Measurements

For the baseline measurements, version `Vct1` of the tool was used. This version was current in June 2018. In Table 3.1 we show the verification times of the examples from the set of representable examples for version `Vct1` of VerCors. In the second column, there are the number of runs that were done to compute

File	#	Verdict	Total time		Back-end time	
			mean	stdev	mean	stdev
case-studies/prefixsum-drf.pvl	5	Pass	193286	6601	185645	6567
witnesses/TreeWandSilver.java	5	Pass	33340	214	30495	276
carp/summation-kernel-1.pvl	5	Pass	24320	210	16726	121
verifythis2018/challenge2.pvl	5	Pass	21709	186	13346	59
floats/TestHist.java	5	Pass	21423	247	16021	261
carp/histogram-submatrix.c	5	Pass	19579	252	11901	30
floats/TestFloat.java	5	Pass	15827	488	9195	143
openmp/add-spec-simd.c	5	Pass	14440	433	7694	82
layers/LFQHist.java	5	Pass	13597	397	8482	175
openmp/addvec2.pvl	5	Pass	12822	144	6612	73
arrays/DutchNationalFlag.pvl	5	Pass	10799	98	5900	168
futures/TestFuture.pvl	5	Pass	6824	202	3955	55
manual/option.pvl	5	Pass	6798	226	2994	126
waitnotify/Queue.pvl	5	Pass	5441	102	3366	102
type-casts/TypeExample1.java	5	Pass	5261	182	3230	64
basic/CollectionTest.pvl	5	Pass	5047	70	3200	40

Table 3.1: Verification times (in ms) for VerCors version `Vct1`.

the times. For each time the mean and the standard deviation is given. With a deviation of the total time that is less than 4% of the total time for all the examples, we argue that 5 runs are sufficient to get an accurate enough indication of the verification times. As we can see, for examples that take more time to verify relatively more time is spent in the back-end. This indicates that the transformation process in VerCors itself does not cause the performance bottlenecks for these examples.

Details about the system that was used for the measurements in this thesis can be found in Appendix C.

3.3 Learn verification times for AST node types

In order to get an idea of what kind of concurrency abstractions take the most time to verify, a framework was developed to measure this. As explained in Section 2.3, VerCors translates the input program to the COL language. This language is defined by an abstract syntax tree (AST) that contains nodes that represent each type of abstraction. E.g. parallel regions, parallel blocks, barriers, etc. The nodes that represent the high-level abstractions are then transformed to equivalent AST constructs consisting of "simpler" node-types. This way all high-level abstractions are reduced to an AST that can be expressed in the Viper language. By visiting all nodes in the COL AST before it is transformed, we can count how many nodes of a specific type there are in the program. After verification we know how long the verification of the program took. We

can divide the time by the number of expressions we counted. This time we call the 'unit time'. For all node types we store a value called the 'learned unit time'. Every time a node-type occurs in a verification run, the value for this node type is adjusted to the unit time. The node-types that represent abstractions that are slower to verify will thus end up with a higher learned unit time. This gives an indication to which transformations might require our attention for performance improvements. Since the unit times per node-type for all verification runs are reduced to one value, the latest verification run will have the most impact on this value. To reduce the deviation the value is updated with a fraction of the new unit time: newValue = (1−f)∗ oldValue + f ∗ unitTime. Where 'f' is the fraction to update with. To get an idea about how much the unit time for a specific node-type fluctuates, we also keep track of the absolute difference between the new unit time and the old value of the learned unit time. This absolute difference is stored and updated just as the unit times. If the learned absolute difference of a unit time for a specific node-type is low, then the learned unit time is a more accurate indicator of the time it takes to verify the concurrency abstraction that this node-type represents.

For illustration let's consider the following expression, that specifies that all elements in the array named input are zero:

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

In this example we count the following expressions:

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

Thus we count 13 expressions in total. If we assume that verifying this expression would take 2600 ms, we get the unit time of 2600/13 = 200 ms for all the 9 node types in this example. Now the learned unit times for these 9 node types are adjusted to this new unit time. Note that the number of expressions is only used to determine the unit time. A larger program, with more expressions, that would take longer to verify, would thus not per see have a greater unit time for the specific node types. Thereby the unit time is a normalized value for a type of AST node.

There might also be special AST constructs of which we want to find out if they take more time to verify. For this purpose, a generic SpecialCountVisitor

has been implemented, that walks the syntax tree. By overriding visit methods, and increasing a counter if a node matches the AST construct, we can keep track on how many times these constructs occur in the AST. The counts of these special constructs will not add to the count that determines the unit time for a given verification run, since they are made up of nodes that are already counted. However the entries in the 'learned unit times' file for these special constructs are adjusted to the unit time of the verification run. Again, this allows us to see if these constructs are slower to verify in general. For instance, the TypesCountVisitor is such a visitor. This visitor keeps track of which data types are used in the input program. This allows us to determine if the encoding of certain data types cause poor performance in verification.

The results with the learned unit times are stored in the hidden folder `.learn/`. All the files are in JSON format. Therefore they can be read with a regular text editor. The entries in the files are sorted by descending values, thereby making it easier to spot which entries need our attention. Since there are multiple results we want to distinguish, we adopted the following naming schema:

- The filename starts with the name of the back-end that is being used followed by an underscore.
- Next in the filename is the name that the developer gave to the counting pass. For now this is only the "before_rewrite" name, since that is where the pass takes place. In case developers want to monitor unit times at different points in the transformation process of VerCors, additional counting passes with a different name can be added to the list of compiler passes. These counting passes will then be stored in a separate file with the given name.
- The values that monitor how much the unit times differ are stored in a separate file. For this file the name is appended with "_diff". Thus if a developer adds an additional counting pass, an additional file in which the unit time differences are monitored is automatically generated.

Learning unit times for AST nodes is an analysis tool, and is not required to perform program verification. Therefore this feature is not enabled by default. In order to use this feature VerCors must be called with the `--learn` command-line option.

If we run the test suite of VerCors on the entire example directory we get the learned unit times from Appendix C.1.

3.4 Discussion

One of the most prevalent examples that was slow to verify, suggested by users, is the parallel prefixsum example in the VerCors example repository [16]. Analyzing verification times for the other examples in the VerCors example repository also showed that the prefixsum example is the slowest to verify. This example

has thus been used to identify bottlenecks. Other indications that users gave were not very specific, but have been investigated: The use of quantifiers seem to make verification slow, and specifications with (quantified) permissions also seem to take more time. In Chapter 5, we look more closely at quantifiers.

Looking at the Viper translation that VerCors generated for the prefixsum example, one of the experts identified the way VerCors encodes arrays as being a potential bottleneck. Therefore, the array encoding is investigated in more detail in section 4.

The results in Section 3.2 show that, for examples that take longer to verify, most of the time is spent in the back-end verifier. This indicates that the transformation process in VerCors is not the cause of a performance bottleneck for these examples.

The unit times in Appendix C.1 shows that the encoding of Strings and Options seem to be slow. For the Option type, we can also see that the learned difference in unit time is 0.0. This is because there is only one example that uses Options (before rewriting). Further we see that the operations `ValidMatrix` and `ValidArray` have high unit times, just as the `BindingExpression` with the `Star` binder, which represents a universal quantifier to specify permissions on (un)bounded heap structures. We know that these AST structures are frequent in the parallel prefixsum example. This corresponds to the information given by the users and Viper experts.

Running a limited set of examples gives us only a minimal indication of what is going on. Since the AST does not change between runs, and verification times will also be similar each time. Therefore learning is best done in an environment that is used for many different verification problems, such as an online demo environment where lots of different programs are verified.

The framework can be extended by adding AST visitors to recognize and count other program constructs. This makes it possible to adapt the implementation to monitor other program constructs, if suspicions arise that those constructs may cause poor performance.

Chapter 4

Array encoding

In this chapter, we elaborate on how array structures are treated in VerCors. As discussed in Chapter 3 the encoding of arrays, and more specifically the encoding of injectivity of arrays, was identified as being a potential bottleneck in verification. In this chapter, we look at an alternative encoding of arrays. First, in Section 4.1, we look at how arrays were encoded in the past, the reasons why it was encoded this way and the shortcomings of this encoding. Next we look at a new encoding in Section 4.2. In Section 4.3 we compare verification results for versions of the tool with the old and the new array encoding. Finally we discuss the results in Section 4.4.

4.1 Old array encoding

In the parsing phase, all Java, C and PVL arrays are transformed to COL arrays. During the COL transformations, the Array type is wrapped in an Option type. In this way we can reason about initialized and uninitialized arrays. An OptionNone models an uninitialized array (a C array pointing to NULL or a Java or PVL array being null), while an OptionSome contains the initialized array. In order to reason about permissions to elements in the array, the inner type of the array is wrapped in a Cell type. This Cell type is in a later transformation transformed to a reference to a field in Viper, which allows us to reason about permissions on this field. This transformation is implemented in the "rewrite_arrays" compiler pass. Thus the transformation chain for the old array encoding looks as follows:

Input type	Type after parsing	rewrite_arrays
int []	Array< Integer>	Option<Seq<Cell<Integer>>>

Multi-dimensional arrays (arrays of arrays) were flattened to a one-dimensional array. An additional function was generated by VerCors to calculate the index of the element in the flattened array that corresponds to the element of the multi-dimensional array. This was done because previous versions of Viper did not allow nested quantification, or quantification with multiple variables, for permissions. This made it hard to reason about permissions on elements in multi-dimensional arrays. This additional transformation is done in the "recognize_multidim" compiler pass.

To illustrate what changes in the "recognize_multidim" transformation we use the code in Listing 4.1. In this example, the type of the multi-dimensional array "matrix" would change from Array<Array<Integer>> with the dimensions M and N, to an Array<Integer> with length $M * N$. The assignment to matrix[i][j] would then change to an assignment to matrix[multidim_index_2(M, N, i, j)]. In the later "rewrite_arrays" transformation, the types would then change as if it is a one-dimensional array. Note that it is necessary to specify the dimensions of the arrays, since these are needed to calculate the corresponding index for the flattened array.

Listing 4.1: Multi-dimensional array access.

```
void setVal(int[M][N] matrix, int i, int j, int val) {
  matrix[i][j] = val;
}
```

For completeness, the Viper code for the multidim_2 function is given below. The additional postconditions of the function are needed to specify that every tuple of indices (in this case the 2-tuple (i0, i1)) is an injective mapping to the new index as calculated by the function (provided that the indices are within the bounds as specified in the preconditions). The function to calculate the new index is a non-linear function. Since non-linear theories in general are undecidable, the contract of this function can not always be proven. Therefore we need to assume the post conditions of the function. In Viper this can be done by omitting the body of the function. Without a body of the function to verify, Viper will assume the postconditions after the preconditions have been checked.

```
1  function multidim_index_2(N0: Int, N1: Int, i0: Int, i1:
       Int): Int
2    requires 0 <= i0
3    requires i0 < N0
4    requires 0 <= N0
5    requires 0 <= i1
6    requires i1 < N1
7    requires 0 <= N1
8    ensures 0 <= result
9    ensures result < N0 * N1
10   ensures result == i0 * N1 + i1
11   ensures result % N1 == i1
```

In older versions of Viper, there was no check on bounds in sequences. However later versions of Viper (the versions of Viper as used from `Vct2` onwards in VerCors)also require that all accesses to a sequence are within bounds. This is where the non-linear calculation as used for multi-dimensional arrays becomes a problem. In Viper all integer arithmetic is passed to Z3. Unfortunately Z3 can not reliably discharge proof obligations about non-linear arithmetic. Therefore Viper can not reliably prove that an access to a sequence using the multidim_X function (where X is the number of dimensions of the array) is within bounds, and will therefore fail on this point.

By construction, arrays in Java and C are injective. That means that every element in the array is different from all other elements in the array (every slot is a distinct block of memory). With the encoding as a sequence of references, this was not necessarily the case in VerCors, since a sequence could look like xs = 0X4, 0X4, 0X8, ..., in which the first two elements are references to the same block of memory. Therefore in VerCors there was the possibility to specify so-called valid arrays and valid matrices, with the keywords `\array(<name>, <dim>)` and `\matrix(<name>, <dim1>, <dim2>)`. Being valid means that the array or matrix is not null, has the specified dimensions and that all slots are different. Since the encoding of this property was identified as being a possible bottleneck in verification, we will look at this encoding more closely. The specification `\array(input, N)` is translated to the Viper code as shown in Listing 4.2.

Listing 4.2: Viper encoding of `\array(input, N)`.

```
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))
```

For a matrix the result would look similar, since the matrix is flattened to a one-dimensional array. Only the upper bounds on i and j would change to the product of the dimensions. In order to reason about actual values in the array that is quantified over, the prover would need to put actual values in place of the i and the j in the example from Listing 4.2. To consider possible values for these variables the prover has to know how the variables are used. This way the prover can match statements in the program with appropriate instances of the body of the quantified expression. Directives to the prover to match instances for quantified expressions to actual values are called triggers. In Viper there is the option to supply triggers for quantified expressions. If no triggers are specified, Viper will try to infer suitable triggers. However, VerCors does not support triggers in the input languages, since additional transformations on the AST may render the triggers unsuitable. Unsuitable triggers can cause the prover to fail to create the right instances to proof or disproof a verification condition.

Therefore, VerCors relies entirely on the triggers that are inferred by Viper. We refer to the article about Simplify by Detlefs et al [8], for an in depth explanation on how triggers can help the back-end provers to guide quantifier instantiations to establish a proof. This is the technique as used in Z3 [7], on which the Viper tool builds. In the body of the quantified expression are two array accesses. One for the element at index i and one for the element at index j. To prove this predicate, Viper would infer two triggers for the expression, namely a trigger to match on the access to element i and one to match the access of element j. This means that for every element of the array two instances are created, thereby creating a possible quadratic blowup of relations that the prover has to maintain.

Note however, that it is often unnecessary to specify array injectivity explicitly. If the specification already requires more than a half permission on every element in the array, this implicitly means that every element is different. Since if two elements would have the same memory location we would require a permission larger than 1 on that location (two times a permission greater than half), which is not possible. Only when we require read permissions smaller or equal to 1/2 on the elements in the array, it might thus be necessary to specify injectivity explicitly.

In order to address the problem that VerCors can no longer reliably reason about multi-dimensional arrays using the newer versions of Viper, and to reason more efficiently about array injectivity, we will look at an alternative encoding of arrays in the next section.

4.2 New array encoding

In recent versions of Viper it is possible to use universal quantifiers for expressions with multiple quantified variables. Therefore, it is no longer required to flatten multi-dimensional arrays. This also means that the non-linear arithmetic to calculate the index into the flattened array is no longer required, giving us a possibility to reinstate the support for multi-dimensional arrays.

We created the domain encoding in Listing 4.3 to model the injectivity of arrays. In this domain 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 Ref type, 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 , which encodes injectivity. Note that the axiom all_diff has only one trigger (loc(a, i)), compared to the two triggers that were inferred in the old array encoding to specify injectivity. Further there is the function alen, to model the length of the array.

Listing 4.3: Viper domain to encode 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
  }
}
```

Because multi-dimensional arrays are no longer flattened we must now also consider the inner arrays within the multi-dimensional array. For clarity we will use a 2-dimensional matrix to talk about the multi-dimensional array, and we will use a row to refer to an inner array within the matrix. Just as in the old encoding we wrap the outer array (matrix) in an Option type. In this way we can 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, and variants where rows can not be NULL. It is clear that these semantics are opposite, and we can not support them all in the way they are transformed currently. In Section 4.4.1 we will look more closely at these variants and their support that is currently under development in VerCors. For now, the choice was made to also wrap the rows in Option types, because this gives a more consistent type transformation within the COL language. Since it has never been possible to reason about permissions or values of entire rows in a matrix by VerCors, the rows are not wrapped in a Cell type. The inner types of the rows are being wrapped in a Cell type, because we do want to be able to reason about permissions to the elements of the matrix. This gives us the following transformations for a 2-dimensional matrix:

Input type	Type after parsing	rewrite_arrays
int [][] matrix	Array<Array<Integer>> matrix	Option<Array<Option <Array<Cell<Integer >>>>> matrix
matrix[i][j]	matrix[i][j]	OptionGet(OptionGet(matrix)[i])[j].item

Note that in a later transformation VerCors will generate the not-null requirements from the OptionGet operation, and a dereference to a field of type Integer from the .item dereference.

With this new encoding, it is no longer required to explicitly specify injectivity for valid arrays and matrices. Injectivity for arrays is implicitly encoded by the all_diff axiom in the VCTArray domain, and for a matrix like the one in the example, Viper can also deduce that every row in the matrix is a different row. Therefore the specifications that are generated for a valid array now only contain a not-null condition and that the array has the specified length. For a valid matrix a not-null condition is generated for the outer array, and that the matrix has the specified number of rows. Additionally for every row in the matrix it will also generate a not-null constraint and that the row has the specified number of cells (columns). 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<Array<Cell <Integer>>> type, so that two elements of the outer array of the matrix could point to the same Ref. In the implementation we have already anticipated on the possibility that VerCors will have to support multiple variants of the valid matrix annotation, for the cases where rows might be objects (like in Java) or when rows are consecutive blocks of memory (like in some variants in C). The appropriate constraints are generated based on the types, which for now are determined by the rewrite_arrays transformation, but which will in the future be determined by the front-end based on the type of program that is being verified.

4.3 Results

To compare the performance of verification in Vercors with the old and the new encoding of arrays, we made use of the scripts and the set of representable examples as described in section 3.2. As noted before, the examples with multi-dimensional arrays are broken in the current version of Vercors, due to changes in the Viper back-end. Therefore we have compared results for three versions of the tool.

- `Vct1`, where examples with multi-dimensional arrays were still working.
- `Vct2`. At this point in time development of the new encoding of arrays was started. In this version part of the support for multi-dimensional arrays in VerCors was already broken, due to changes to the Viper back-end.
- `Vct3`, which is a more recent version of VerCors. This version again has a newer version of the Viper back-end.

Then we have two versions with the new array encoding: `Vct2-a`, which is the new array encoding as developed on top of the `Vct2` version and `Vct3-a`, which is the `Vct2-a` version updated to the `Vct3` version of VerCors. Figure 4.1 shows a schematic overview of the source tree. References to the exact points in the Git history can be found in Appendix A.

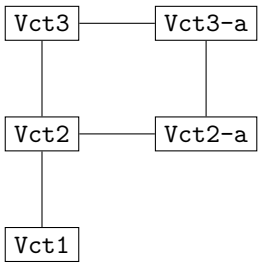

Figure 4.1: VerCors project source tree (array encoding)

To get accurate results we used the scripts from section 3.2, to verify each example from the set of representable examples 5 times, for each version of the tool. Since a number of examples were broken in the `Vct2` and `Vct3` version of the tool, we first compare the results for the `Vct1` and `Vct2` version of the tool to the `Vct2-a` version. Then we also compare the `Vct2` and `Vct3` version to the `Vct3-a` version. Note that failing examples are marked with a "-".

In the set of representable examples, the following examples make use of arrays:

- `case-studies/prefixsum-drf.pvl`
- `verifythis2018/challenge2.pvl`
- `arrays/DutchNationalFlag.pvl`
- `openmp/addvec2.pvl`
- `carp/summation-kernel-1.pvl`

Of which the `case-studies/prefixsum-drf.pvl` is the only example that makes use of a multi-dimensional array.

4.4 Discussion

As we can see in Table 4.1, the `carp/histogram-submatrix.c` example and the `case-studies/prefixsum-drf.pvl` fail to verify in version `Vct2`. This is due to the changes in Viper, which cause examples with multi-dimensional arrays to break. We also see that the `verifythis2018/Challenge2.pvl` example fails for version `Vct2` and `Vct2-a`. On closer inspection we found that this is not due to the array encoding. This example sometimes gets a Pass verdict, and sometimes a Fail verdict. When the example fails VerCors reports "⟨No location⟩". However on further investigation it seems that the failure is in a loop invariant, which has nothing to do with arrays. Therefore we believe that this inconsistent behavior is caused by something else. In the list with representable examples, the `case-studies/prefixsum-drf.pvl` example has three invariants with the `\array` and `\matrix` annotations. Table 4.1 shows that this example

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

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

is nearly four times faster to verify in the `Vct2-a` version compared to the `Vct1` version. We also see that some examples are slower to verify in the `Vct2-a` version compared to the `Vct2` version. We believe this might be because the sequences that are used in the old encoding (`Vct2`) are a primitive type in Viper. Therefore, optimizations in Viper might allow Viper to reason more efficiently about elements in sequences, whereas in the new encoding (`Vct2-a`) Viper has to maintain the additional relation between the VCTArray object and the index that define an element in the array.

Table 4.2 shows the effect of the new encoding in relation to the newer version of VerCors with the old encoding. We see that in the `Vct3` and the `Vct3-a` version the `floats/TestFloat.java` example fails. This verification example does not terminate due to a matching loop in a universal quantifier. Matchings for quantifier instantiations in general influence verification performance. We will look at this in detail in chapter 5.
As we can see the `Vct3` version of VerCors overall performs worse than the `Vct2` version. This negative effect seems to be reinforced by the new array encoding in the `Vct3-a` version.

In the new encoding injectivity of an array no longer has to be specified explicitly. Being able to infer this property automatically significantly improved performance in cases where this property is required. This is mostly the case when only read permissions are required on the elements of an array. Also, it is no longer required to flatten multi-dimensional arrays to one-dimensional arrays. This eliminates the need for the non-linear arithmetic required to determine the (injective) relation between indices in the multi-dimensional array and the (calculated) corresponding index in the flattened array.

In the old situation there was no explicit check on the bounds of an array. This is also the case for the new encoding. Since every heap location, thus every element in the array, is protected by permissions this does not have to be an issue. If the permissions on the elements of the array are specified correctly, it is not possible to read or write out of bounds, since the thread would not have permission to read or write to that location. It might, however, be desirable to add explicit bound checks, to protect users from inadvertently writing erroneous specifications. This could be done by generating a wrapper function for loc(a: VCTArray, i: Integer) with the precondition that 0<=i && i<alen(a) (in other words, i is within bounds of the array).

4.4.1 Array differences

As noted before VerCors translates both Java and C arrays the same way. They are initially parsed to the Array type in COL. In a later transformation they are wrapped in an Option type and the inner elements are wrapped in a Cell type. However different sorts of arrays have different semantics in Java and C. Table 4.3 shows the various types of arrays in Java and C, their semantics, and if they are supported in VerCors. The column "Nullable" indicates whether a matrix (or array) can be `null` or `NULL`. The column "Row assignable" indicates whether or not another array can be assigned to a row of the matrix. In this list we also

Lang	Decl	Nullable	Row assignable	VerCors support
Java	**int** [][] matrix;	Yes	Yes	No
	int [] matrix [];	Yes	Yes	Yes
	int matrix [][];	Yes	Yes	No
C	**int** matrix[M][N];	No	No	Yes[1]
	int* matrix[N];	No	No	No
	int ** matrix;	Yes	Yes	No
PVL	int [][] matrix;	Yes	Yes	Yes

Table 4.3: Valid array types

look at pointers in C, since for many concurrent programs this mechanism is also used to work with subsequent blocks of memory. For PVL we only included the notation that is supported by VerCors, since there is no exact specification of the language.

As it can be seen, not all array declaration notations are supported by VerCors. Further we see that the semantics differ between Java arrays and C arrays, and that C pointers have different semantics than C arrays. To better support the differences between the different types of arrays, the transformation should not be done in the COL language, since we would like the COL language to be consistent independent of the input language. Therefore the transformations whether or not arrays should be wrapped in an Option type, and whether the elements within the array should be wrapped in a Cell type, should be taken care of by the pre-COL transformation phase. This is mostly an engineering task, that is outside of the scope of this thesis. Currently work has started to integrate these improvements, however they are not yet merged into the tool. The transformations that generate the correct code for the `\array` and `\matrix` annotations, as well as the code that is generated to model array constructors, is prepared so that it will generate the correct code based on the types that are provided.

[1] Assigning to rows is wrongfully accepted by VerCors

Chapter 5

Trigger generation

In this chapter we look at another program structure that was identified as being slow to verify. This concerns universal quantifiers that are used to specify properties over possibly unbounded data structures. First we describe how quantified expressions are treated in VerCors and how they are passed to the Viper back-end verifier in Section 5.1. We describe how triggers can help the solver to discharge a proposition with universal quantifiers more quickly. Then we discuss the transformations that are needed to be able to generate valid triggers in Section 5.2. In Section 5.3 and 5.4 we describe how triggers are generated and how this is implemented in the tool. We compare performance of verification for versions with and without triggers in Section 5.5, and we discuss the results in Section 5.6.

5.1 The structure of quantifiers

One of the ways VerCors supports reasoning over unbounded data structures is by the use of quantifiers. The notation of the quantifiers is similar to the JML syntax, with an additional `forall*` to specify universal quantification over permissions. In the COL language the four main components of a quantified expression are structured as follows:

- Binder: to specify the type of quantifier.
- Declarations: to declare the variables that are quantified over.
- Selection: to specify the domain of the quantification.
- Main: the the body to which all elements in the domain have to adhere.

For illustration let's consider the following JML statement:

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

This statement specifies that all elements of the array called "input" have the value 0. In this statement the binder type is `Forall`. The single declaration

is **int** i, and the selection is 0<=i && i<input.length. The main is the equality input[i] == 0. In the Viper language there is no distinction between the selection and the main part of the quantifier. Therefore the selection and main are rewritten into an implication. This looks as follows: \forall **int** i; 0<=i && i<input.length; input[i] == 0; → forall i: Int :: 0<=i && i<|input| ==> input[i].Integer__item == 0.

Triggers

In order for an SMT solver to reason about these propositions it has to instantiate the body of the quantifier with actual values in place of the quantified variables. But since this can be any value, it is important that the solver only uses instantiations that are relevant for the problem. This means that it should make the right instantiation to discharge a proof, but preferably not more, since adding irrelevant knowledge makes solving unnecessary slower. The SMT solver that is used in Viper, namely Z3, can be guided to trigger the right quantifier instantiations. This is done by supplying so-called triggers, which are patterns that the SMT solver can use to match against. If the solver encounters a statement with the given pattern(s), it will instantiate the quantifiers body with values for the current context. In the Viper language a user can also specify sets of triggers for the universal quantifier expressions, which are then used to guide Z3 into making the right quantifier instantiations. There are a couple of restrictions that Viper imposes on triggers, namely:

- Each quantified variable must occur at least once in a set of triggers.
- Each trigger must contain at least one quantified variable.
- Each trigger must have some sort of structure, e.g. a function application. A quantified variable itself is not a valid trigger.
- Triggers may not contain arithmetic or boolean operators.
- Accessibility predicates (permissions) are not allowed in trigger expressions.

In short these restrictions stem from the way relations are canonicalized in the SMT solver. For an in depth explanation we refer to the article on Simplify, by Detlefs et al [8].

If the user does not supply triggers, Viper will try to infer them from the body of the quantified expression. A downside of letting Viper infer the triggers is that it will most likely infer too many triggers. For instance let's consider the Viper program in Listing 5.1 from the Viper tutorial[1]. For the quantified expression in the axiom `axsum`, Viper can infer the following trigger sets:

- {sum(a, b)}

[1]The Viper tutorial is located at `http://viper.ethz.ch/tutorial`

- { get_value(a), get_value(b) }
- { sum(a,b) }{ get_value(a), get_value(b) }
- { get_value(a), get_value(b) }{ sum(a,b) }

Thus for every state in which a match for all the triggers in one of the trigger sets is found, an instance of the quantified expression is generated. However, intuitively it is clear that we only need the axiom `axsum` to be applied when the pattern sum(a, b) is encountered. All other trigger combinations will lead to unnecessary quantifier instantiations, making verification slower. Therefore it is beneficial to supply the right triggers.

Listing 5.1: Viper specification for an Integer domain.

```
1  domain MyInteger {
2    function create_int(x: Int): MyInteger
3    function get_value(a: MyInteger): Int
4    function sum(a: MyInteger, b: MyInteger): MyInteger
5
6    axiom axCreate {
7  forall i: Int :: get_value(create_int(i)) == i
8    }
9
10   axiom axSum {
11     forall a: MyInteger, b: MyInteger ::
12       sum(a,b) == create_int( get_value(a) + get_value(b) )
13   }
14 }
```

Matching loops

An extreme effect of poorly chosen triggers (or poorly inferred triggers) is a matching loop. Listing 5.2 shows a specification of a factorial function. One of the candidate expressions to become a trigger for the quantifier at line 8 is the fact(i) term. However, if the solver would instantiate the body of the quantifier for this pattern, it would encounter another match for the term fact(i−1). If the solver would not be able to prove that i eventually is smaller than 1, then the solver would continue generating instances indefinitely. When specifications become more complex, and are composed of multiple functions, it becomes less trivial to deduce where a matching loop could occur. As we have seen in Section 4.3, there is also an example in our set of representable examples that is plagued by this effect. Using the axiom profiler of Becker et al [3], we were able to determine that this example does not terminate because of a matching loop.

Listing 5.2: Specification of a factorial function.

```
1  class Factorial {
```

```
2     ensures \result == (n>1 ? n*fact(n-1): 1);
3     pure int fact(int n) {
4       return n<=1 ? 1 : n*fact(n-1);
5     }
6
7     void main() {
8       assert (\forall int i; 0<i; fact(i) == i * fact(i-1));
9     }
10  }
```

In VerCors there is no support for triggers in the input language. First because the VerCors specification language is based on JML, which does not use triggers. Second because VerCors rewrites the AST, and could thereby invalidate triggers. For instance, if new quantifiers are generated by VerCors to encode parallel blocks, this could add quantifier variables, which are not covered by the triggers that were in the input program. Therefore VerCors relies on Vipers capability of inferring triggers. Because quantifiers were identified as being a possible bottleneck in verification with VerCors, we have investigated if we could generate triggers for Viper and if that would improve performance.

5.2 Rewriting complex subscripts

Data structures that are often used in combination with quantified expressions are arrays or sequences. For these data structures quantifiers are often used to specify properties for all elements in the array or sequence. In the COL language that is used internally in VerCors, accessing a specific element in an array or sequence is modeled by the `Subscript` operator. A statement like input[2], where "input" is an array, would be represented as an `OperatorExpression` with the operator `Subscript` and the arguments `input` and 2. If we encounter such an expression in the body of a quantified expression, and we want to generate a trigger for it, we must adhere to the restrictions that Viper imposes on triggers. For instance, we can not use a trigger of the form input[i+1], since the subscript part of the expression uses the + operator. One possibility is to try to rewrite these kinds of subscripts to a form that is a valid trigger expression. To illustrate this procedure, let's consider the example in Listing 5.3.

Listing 5.3: Function specification with complex subscript.

```
1  class Subscripts {
2    invariant (\forall int i; 0<=i && i<|s|/2; s[i] == s[2*
         i]);
3    void fun(seq<int> s);
4  }
```

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 expression there are two candidates for

trigger expressions: s[i] and s[2∗i]. However, this is not a valid trigger set, since the second expression has the arithmetic operator ∗. What we can do to eliminate this multiplication is to introduce a fresh quantifier variable. Then we can replace the $2 * i$ expression in the body of the expression with the new variable, and we add an equality to the selection of the expression that the new variable should be equal to $2 * i$. That gives us the following expression:

(\forall int i,int unit_var_1; 0<=i && i<|s|/2 && unit_var_1 == 2∗i; s[i] == s[unit_var_1]).

This gives us the candidate triggers s[i] and s[unit_var_1]. As a set, these triggers do adhere to the restrictions that Viper imposes on triggers:

- Both quantified variables occur in the set of triggers.
- Each trigger contains a quantified variable.
- Each trigger has a structure (in this case a subscript operation).
- There is no longer an arithmetic operator in (one of the) triggers.
- There are no accessibility predicates in the triggers.

One aspect that we do need to consider, 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. That could make it even harder for the SMT solver to find the right instances to discharge a proof. Our hypothesis, however, is that the SMT solver can easily discharge the added equality in the selection of the quantifier (in our example the equality unit_var_1 == 2∗i), and that the positive effect of being able to generate an appropriate trigger will outweigh the costs of the added complexity.

5.3 Generating triggers

After eliminating complex subscripts we can try to generate trigger sets for universal quantifiers, at the end of the rewriting phase. This ensures that the generated triggers will not be invalidated by other transformations. Since the selection and the main elements of a COL quantifier are rewritten to an implication in the body of the Viper quantifier, we have to look for trigger candidates in the selection and the main of the COL quantifier. These trigger candidates have to adhere to the restrictions that Viper imposes on triggers. A trigger candidate is thus an expression that mentions at least one of the quantified variables, has some sort of structure (thus is not the variable itself) and does not contain an accessibility predicate. From this set of trigger candidates we compose valid trigger sets. We do this by first generating the powerset of the set with triggers (excluding the empty set, since the empty set is not a valid trigger set). From this powerset we select all the sets that mention all quantified variables, and are thereby valid trigger sets. 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.

5.4 Implementation

To enable the rewriting of complex subscripts in quantifiers and adding triggers, the optional command line flag `--triggers` has been introduced. This command line flag should receive an integer value. Internally, the binary representation of the value is used to switch on or off the rewriting of complex subscripts and trigger generation. If the second least bit is set (value & 2), the complex subscripts are rewritten. If the least significant bit is set (value & 1), then triggers are generated. This gives us the following possible values:

0 No additional transformation will be made.

1 Will add triggers if possible, without any further transformations.

2 Will rewrite complex subscripts to a new quantified variable, and will not add triggers.

3 Will rewrite complex subscripts and add triggers if possible.

5.5 Results

To check the effect of rewriting complex subscripts and adding triggers to quantifiers, we compared verification times for the set of representable examples from Section 3.2. We compare verification times for version `Vct3` of the tool, for all four options for the `--trigger` flag: No additional transformations (`Vct3`), only add triggers (`Vct3-t1`), only rewrite complex subscripts in quantifiers (`Vct3-t2`) and rewriting complex subscrips in quantifiers plus adding triggers (`Vct3-t3`). For each version the example is verified 5 times, and the average is shown in Table 5.1.

However from Chapter 4 we know that examples using multi-dimensional arrays could not be verified with version `Vct3` of the tool. Therefore we have also applied our changes to implement the triggers features to the `Vct3-a` version of the tool. This gives us the versions `Vct3-a-t1` (`Vct3-a` plus trigger generation), `Vct3-a-t2` (`Vct3-a` plus rewriting complex subscripts in quantifiers) and `Vct3-a-t3` (`Vct3-a` plus rewriting complex subscripts and trigger generation). In Table 5.2 are the verification times for the examples using this version.

References to the exact points in the Git history for version `Vct3-t` and `Vct3-a-t` can be found in Appendix A.

File	Vct3	Vct3-t1	Vct3-t2	Vct3-t3
floats/TestFloat.java	-	17483	-	17516
openmp/add-spec-simd.c	15514	15646	-	-
openmp/addvec2.pvl	14673	14857	-	-
basic/CollectionTest.pvl	8483	8429	8400	8395
futures/TestFuture.pvl	9568	9730	9558	9492
manual/option.pvl	10275	10352	10299	10202
floats/TestHist.java	20226	19997	20297	20212
arrays/DutchNationalFlag.pvl	13887	13913	13870	13951
waitnotify/Queue.pvl	8283	8458	8504	8340
carp/summation-kernel-1.pvl	19694	19770	19817	19865
type-casts/TypeExample1.java	8022	8161	8144	8152
layers/LFQHist.java	16130	16717	16398	16441
witnesses/TreeWandSilver.java	29072	29841	30250	29682

Table 5.1: Comparing total verification times (in ms) for version Vct3, with and without triggers. Sorted by relative speedup from Vct3 to Vct3-t3.

5.6 Discussion

From the results in Table 5.1 we can note multiple things. First of all we see that in the TestFloat.java example the added triggers eliminate the matching loop that caused the verification to run indefinitely. In the next two examples we see that the complexity that is introduced by rewriting complex subscripts to additional quantified variables cause the verification to fail. Further we see that generating triggers does not have a positive effect on verification times for these examples. Next we see that the Challenge2.pvl only succeeds for version Vct3. However in Section 4.3 we already concluded that verification of this example is inconsistent. Therefore we can not base conclusions on this result. For the remaining examples we see that the added transformations are of a minimal effect on the total verification times.

For the results with the new array encoding in combination with generating triggers from Table 5.2, we can note the same for the TestFloat.java example, namely that the added triggers help break the matching loop that caused this example to run indefinitely. Then there are the same two examples that fail to verify when complex subscripts are rewritten to additional quantified variables. But there also are two examples that benefit significantly from the added transformations: The TestHist.java example takes only 67 % of the time to verify if we rewrite complex subscripts and add triggers, And the prefixsum-drf.pvl example is also 29 % faster when we rewrite complex subscripts and add triggers. For the remaining examples we see that again the effect on total verification time is minimal.

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 5.2: 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`.

Because the effect of rewriting complex subscripts in quantifiers and/ or adding triggers can have a positive or negative effect depending on the context, we chose to not enable this feature by default. Only the `TestFloat.java` example does not verify in the `Vct3` version, and can be verified when triggers are generated. However we think that the matching loop is caused by a more fundamental problem, that needs to be solved in the VCTFloat domain that is used by VerCors to model floating point numbers. For the other examples, rewriting complex subscripts and generating triggers does not necessarily have a positive effect on the performance. Therefore we have kept the `--trigger` option with the integer value to specify what transformations the user wants to apply. In this way expert users that think they can benefit of these features can still use them, but for users unaware of these features the safest way is to not use them by default.

Chapter 6

Related work

This research is focused very specifically at the performance of program verification with VerCors. In literature we can look more broadly at research and comparison of performance between different verification techniques. Cassios et al have published an experience report in which Symbolic Execution (SE) is compared to Verification Condition Generation (VNG) [11]. However, to the best of our knowledge, there is no report on the analysis of specifically which parts of a program have the most influence on the performance of verification.

Leino et al identify matching loops in quantified expressions as a significant contributor to instabilities in performance and user experience in program verification [13]. They propose to move trigger logic away from the SMT solver and into the high level verifier. The paper presents three techniques for trigger selection that are implemented in the Dafny verifier: Quantifier splitting, trigger sharing and matching loop detection. First terms that could be part of a trigger are collected. Then it enumerates subsets of these terms and rejects terms that are not allowed. The trigger candidates are evaluated to see if they can cause a matching loop. If this is the case, then the trigger is also removed from the Pool. In practice users tend to collect related conditions under a single quantifier. However, the technique to prevent matching loops may cause essential triggers to be eliminated if the conditions in the quantifier are only weakly related. To mitigate this problem, the quantifier is split into multiple quantifiers. Thereby preventing the case that a possible matching loop for one condition eliminates an essential trigger for another condition. To recover the triggers that were eliminated by the matching loop detection for the quantifier that is split, the pool of triggers is enriched with the trigger candidates that derive from the same split quantifier. This is what they call trigger sharing. The techniques are implemented in the Dafny verification tool. They show significant performance gains on Dafny's test suite and larger verification problems.

Chapter 7

Conclusion

In this research we wanted to investigate what performance bottlenecks in program verification with VerCors were, in order to optimize performance of the tool. First we have identified two program constructs that could cause poor performance in verification, namely the encoding of (injectivity of) arrays and the use of universal quantifiers without triggers. We have also developed a framework that can help identify which program constructs take relatively more time to verify. First results have shown that the framework can identify the constructs that were also identified by experts as possibly being problematic. We think this framework can help identify new problematic constructs as the tool evolves. If the encoding of existing concurrency abstractions in VerCors changes, the learned normalized verification times for the nodes that model this abstraction will automatically change accordingly. Thus if the change has a negative effect on performance, this will show up in higher verification times. Further, if new abstractions are added to the tool, for instance by adding new node types in the COL AST, then they can easily be added to the framework. This makes it possible to immediately monitor how the performance of verification of this abstraction relates to verification of the rest of the programs.

Next we have developed a new encoding for arrays. Before this research the support for multi-dimensional arrays was broken in VerCors, due to an update in the back-end verifier. Therefore we had to compare results to an older version of VerCors. We have seen a verification speedup of 4 times in cases where array injectivity was a relevant property.

Finally we have investigated if we could benefit from generating trigger sets for universal quantifiers in VerCors. We have seen that in specific cases adding triggers can improve performance. However additional transformations that are needed to be able to generate valid triggers can have a negative effect.

7.1 Future work

In order to improve the framework to monitor normalized verification times for AST nodes, future work could focus on also considering the structure of the syntax tree. For instance taking into account how deep (invariants for) loops are nested, or if certain expressions are used in special contexts.

In the current situation, the learning of normalized verification times for AST nodes is done before the AST is rewritten. This is at the highest level of abstraction. To identify more specifically which part of the transformation causes an abstraction to be slow to verify, it could also be interesting to add counting passes further on in the transformation chain, in order to monitor unit times at different abstraction levels.

With the new array encoding, multi-dimensional arrays no longer have to be flattened to a one-dimensional array. This eliminates the need for a non-linear calculation to map indices from the original array to the flattened array. In VerCors there are rewrite rules that rewrite quantified expressions with subscripts to help the back-end verifier to reason about the non-linear arithmetic. It would be interesting to investigate if eliminating these rewrite rules would have a positive effect on performance.

In transforming abstractions VerCors can generate various quantified expressions. In subsequent transformations these quantifiers might be split and rewritten further. It could be interesting to investigate how these transformations influence the ability to generate valid trigger sets, and how this influences performance.

Currently, the only strategy to select trigger sets for quantified expressions is to generate the powerset of all trigger candidates and filter the valid sets. Future work could focus on how to reduce the set of triggers without eliminating essential trigger expressions.

Bibliography

[1] 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.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

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

[4] 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*, pages 102–110, Cham, 2017. Springer International Publishing.

[5] John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

[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] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. Permission-based separation logic for multithreaded java programs. *arXiv preprint arXiv:1411.0851*, 2014.

[10] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[11] 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.

[12] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[13] 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.

[14] 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.

[15] 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.

[16] UTwente-FMT. VerCors example repository. `https://github.com/utwente-fmt/vercors/tree/master/examples`, 2018. [Online; accessed 24-October-2018].

Appendix A

VerCors repository history

Over time various versions of the VerCors tool have been key in the context of this thesis. With regard to the encoding of arrays these were the points where the old encoding was still working (`Vct1`), the version on which the new encoding is build (`Vct2`) and a more recent version at the time of writing (`Vct3`). The Git commit hashes that mark these points of time in the history of the VerCors repository[1] are the following:

`Vct1` e6f0a2e198ca4e0ccf1fd3e3baae3bb7421cbaec

`Vct2` 50262799fdef5ec65bd396e5d19cac76fd6c07f5

`Vct3` 6f266bacd9e2bb056eff0dddf0fc08ffdca22be8

A.1 Time learning framework

The implementation of the time learning framework as duscussed in Section 3.3 is based on the `Vct1` version of the tool. The Git commit hash that marks this point in the VerCors repository is:

- 7befcbef70fea0425f80092bfaadcc329b514cc0

A.2 New array encoding

The new array encoding, as discussed in Section 4.2, that was based on `Vct2`, can be found in `Vct2-a`. In `Vct3-a` the new array encoding is merged with the later version ov VerCors. The Git commit hashes that mark these points in the VerCors repository are the following:

`Vct2-a` 716a46ad2af31426011287adec1daeecb90d4070

`Vct3-a` eed4c0530d161f269f05ed8dde0026bfb2431d11

[1] The VerCors repository is located at `https://github.com/utwente-fmt/vercors`

A.3 Trigger generation

The implementation for generating triggers, as discussed in Chapter 5, based on version `Vct3` of the VerCors tool, can be found in version `Vct3-t`. In version `Vct3-a-t` the implementation of triggers is merged with the new array encoding from Section 4.2. The Git commit hashes that mark these points in the VerCors repository are:

`Vct3-t` 96922644acb6aa27ab8d4f9d94354f34b159cc0e

`Vct3-a-t` f280dd34aa0ae9cc7fdfdd71eeee761e6d5a0c84

Appendix B

Set of representable examples

Below is the list of examples that is used to compare performance of different versions of the VerCors tool. The examples used are as they were current at commit 7b738ddb45483e43a96f34420c147555e7349f7c.
These examples are all compatible with version `Vct1` of VerCors, and are fixed where possible for the other versions of the tool.

- type-casts/TypeExample1.java
- floats/TestHist.java
- basic/CollectionTest.pvl
- carp/summation-kernel-1.pvl
- floats/TestFloat.java
- openmp/addvec2.pvl
- carp/histogram-submatrix.c
- witnesses/TreeWandSilver.java
- futures/TestFuture.pvl
- case-studies/prefixsum-drf.pvl
- arrays/DutchNationalFlag.pvl
- manual/option.pvl
- waitnotify/Queue.pvl
- layers/LFQHist.java

- openmp/add-spec-simd.c
- verifythis2018/challenge2.pvl

Appendix C

Results

In this appendix are the results as presented in this thesis. Measurements are done with a system with the following specifications:

- Dell Precision M2800
- Processor: Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz, 2501 Mhz, 4 Core(s), 8 Logical Processor(s).
- Memory: 8.00 GB
- Operating system: Windows 10 Pro 1803 X64.

C.1 Normalized verification time per AST node

In this section are the learned unit times as discussed in Section 3.3 and 3.4. In Listing C.1 are the learned unit times. In Listing C.2 is the learned absolute difference for the unit times. The latter gives an indication of the accuracy of the learned unit time.

Listing C.1: Learned unit times for COL AST nodes, before rewriting.

```
{
  "vct.col.ast.PrimitiveType:String": 436.3587529274005,
  "vct.col.ast.BindingExpression:Star:non-linear": 292.2344289092933,
  "vct.col.ast.PrimitiveType:Option": 279.75,
  "vct.col.ast.OperatorExpression:OptionSome": 279.75,
  "vct.col.ast.OperatorExpression:TypeOf": 225.95652173913044,
  "vct.col.ast.TypeVariable": 141.38095238095238,
  "vct.col.ast.OperatorExpression:ValidArray": 130.5266362060251,
  "vct.col.ast.OperatorExpression:ValidMatrix": 127.89861824553546,
  "vct.col.ast.ParallelBarrier": 108.63452278058679,
  "vct.col.ast.OperatorExpression:VectorCompare": 106.82352941176471,
  "vct.col.ast.OperatorExpression:VectorRepeat": 106.82352941176471,
```

```
"vct.col.ast.OperatorExpression:Instance": 95.48913043478261,
"vct.col.ast.PrimitiveType:Set": 94.63157894736842,
"vct.col.ast.PrimitiveType:Bag": 94.63157894736842,
"vct.col.ast.OperatorExpression:Member": 81.92520363408521,
"vct.col.ast.PrimitiveType:Float": 80.74688822959828,
"vct.col.ast.VectorBlock": 77.12446150540876,
"vct.col.ast.OperatorExpression:Cast": 73.5601661721987,
"vct.col.ast.OperatorExpression:PostIncr": 69.90066594039358,
"vct.col.ast.ClassType": 68.94030932345711,
"vct.col.ast.OperatorExpression:PointsTo": 65.97223823682876,
"vct.col.ast.OperatorExpression:Assign": 64.31244197832636,
"vct.col.ast.ConstantExpression": 63.618200185251865,
"vct.col.ast.PrimitiveType:Fraction": 60.83769836833337,
"vct.col.ast.NameExpression": 60.689803585714415,
"vct.col.ast.OperatorExpression:Unfolding": 60.37179255516689,
"vct.col.ast.OperatorExpression:Plus": 59.6403416284453,
"vct.col.ast.PrimitiveType:Integer": 59.48037163183229,
"vct.col.ast.OperatorExpression:Length": 58.70862712081903,
"vct.col.ast.MethodInvokation": 58.59500398341991,
"vct.col.ast.OperatorExpression:Perm": 58.522818043582845,
"vct.col.ast.OperatorExpression:Star": 57.77671580657753,
"vct.col.ast.BindingExpression:Sum": 56.78206715611292,
"vct.col.ast.OperatorExpression:AddAssign": 56.78206715611292,
"vct.col.ast.OperatorExpression:Contribution": 56.78206715611292,
"vct.col.ast.OperatorExpression:ReducibleSum": 56.78206715611292,
"vct.col.ast.OperatorExpression:EQ": 55.43729709091913,
"vct.col.ast.OperatorExpression:GTE": 55.334365677908636,
"vct.col.ast.OperatorExpression:FoldPlus": 54.59464113514258,
"vct.col.ast.OperatorExpression:Old": 54.21488896255912,
"vct.col.ast.OperatorExpression:RangeSeq": 53.20943305500338,
"vct.col.ast.Dereference": 53.16108801103409,
"vct.col.ast.OperatorExpression:ITE": 51.22887005105985,
"vct.col.ast.OperatorExpression:NEQ": 51.19657639001276,
"vct.col.ast.ReturnStatement": 50.94773776639646,
"vct.col.ast.PrimitiveType:Sequence": 50.630117784578324,
"vct.col.ast.LoopStatement": 50.55429828890851,
"vct.col.ast.OperatorExpression:Div": 50.5023980570464,
"vct.col.ast.OperatorExpression:AbstractState": 50.46697590311244,
"vct.col.ast.OperatorExpression:Implies": 49.693318393967175,
"vct.col.ast.IfStatement": 49.582647102894775,
"vct.col.ast.StructValue": 48.36705055979249,
"vct.col.ast.ParallelInvariant": 47.49872322734983,
"vct.col.ast.AssignmentStatement": 47.26844639936432,
"vct.col.ast.OperatorExpression:Append": 46.05298013245033,
"vct.col.ast.OperatorExpression:Drop": 46.05298013245033,
"vct.col.ast.OperatorExpression:NewSilver": 46.05298013245033,
```

```
  "vct.col.ast.OperatorExpression:Take": 46.05298013245033,
  "vct.col.ast.OperatorExpression:Subscript": 45.08085645289825,
  "vct.col.ast.OperatorExpression:Minus": 45.01716715079705,
  "vct.col.ast.BindingExpression:Forall": 44.905261418594286,
  "vct.col.ast.Lemma": 44.29045550669653,
  "vct.col.ast.OperatorExpression:Wand": 44.29045550669653,
  "vct.col.ast.ActionBlock": 43.888575150812656,
  "vct.col.ast.OperatorExpression:LTE": 43.391730315280036,
  "vct.col.ast.OperatorExpression:And": 43.303196355395535,
  "vct.col.ast.OperatorExpression:GT": 43.263177874969806,
  "vct.col.ast.PrimitiveType:Boolean": 43.20991409337897,
  "vct.col.ast.OperatorExpression:LT": 43.09841925678941,
  "vct.col.ast.OperatorExpression:Size": 42.931257949568966,
  "vct.col.ast.ParallelBlock": 41.61108981883007,
  "vct.col.ast.ParallelRegion": 41.61108981883007,
  "vct.col.ast.OperatorExpression:Tail": 41.48513329198524,
  "vct.col.ast.OperatorExpression:Value": 39.74618842614653,
  "vct.col.ast.OperatorExpression:Mult": 38.81302978718812,
  "vct.col.ast.ParallelAtomic": 37.96684230639937,
  "vct.col.ast.PrimitiveType:Array": 37.35537178553868,
  "vct.col.ast.OperatorExpression:MatrixSum": 37.18501170960187,
  "vct.col.ast.BindingExpression:Star": 36.990366990122,
  "vct.col.ast.PrimitiveType:Resource": 36.990366990122,
  "vct.col.ast.OperatorExpression:UMinus": 35.71410305028053,
  "vct.col.ast.OperatorExpression:NewArray": 35.46007883724915,
  "vct.col.ast.OperatorExpression:Future": 35.15452038888792,
  "vct.col.ast.OperatorExpression:Or": 35.12420144337109,
  "vct.col.ast.OperatorExpression:Mod": 30.228637518142236,
  "vct.col.ast.BindingExpression:Forall:non-linear": 29.124944336324056,
  "vct.col.ast.OperatorExpression:HistoryPerm": 29.05927276502691,
  "vct.col.ast.OperatorExpression:Held": 28.557522123893804,
  "vct.col.ast.OperatorExpression:Not": 27.25068405572092,
  "vct.col.ast.OperatorExpression:History": 25.90576401264436,
  "vct.col.ast.PrimitiveType:Process": 25.431439166249206,
  "vct.col.ast.OperatorExpression:Head": 20.90362735911609,
  "vct.col.ast.BindingExpression:Exists": 15.029005524861878,
  "vct.col.ast.OperatorExpression:Scale": 14.542321206709117,
  "vct.col.ast.PrimitiveType:Location": 14.542321206709117
}
```


Listing C.2: Learned differences in unit times for COL AST nodes, before rewriting.


```
{
  "vct.col.ast.PrimitiveType:String": 133.05791373926618,
  "vct.col.ast.ParallelBarrier": 83.18063793071764,
  "vct.col.ast.OperatorExpression:ValidArray": 81.38488910804604,
```

```
"vct.col.ast.OperatorExpression:ValidMatrix": 81.20289476548282,
"vct.col.ast.OperatorExpression:GTE": 77.7256316322934,
"vct.col.ast.BindingExpression:Star:non-linear": 72.64203185865331,
"vct.col.ast.PrimitiveType:Fraction": 53.474205451386744,
"vct.col.ast.OperatorExpression:RangeSeq": 51.04850093794277,
"vct.col.ast.OperatorExpression:Instance": 43.48913043478261,
"vct.col.ast.ReturnStatement": 35.68690977334082,
"vct.col.ast.ParallelBlock": 32.978049904328046,
"vct.col.ast.ParallelRegion": 32.978049904328046,
"vct.col.ast.OperatorExpression:Minus": 32.76574944997405,
"vct.col.ast.ConstantExpression": 31.792495413512757,
"vct.col.ast.ClassType": 30.927997123893483,
"vct.col.ast.PrimitiveType:Array": 30.613146964031394,
"vct.col.ast.OperatorExpression:PointsTo": 30.22668151457074,
"vct.col.ast.OperatorExpression:Subscript": 29.294664627242106,
"vct.col.ast.OperatorExpression:EQ": 29.022056120531673,
"vct.col.ast.BindingExpression:Forall": 28.98793747818828,
"vct.col.ast.OperatorExpression:Size": 28.977205097553295,
"vct.col.ast.ActionBlock": 28.733911609473815,
"vct.col.ast.PrimitiveType:Resource": 28.40931181239899,
"vct.col.ast.BindingExpression:Star": 28.40931181239899,
"vct.col.ast.OperatorExpression:PostIncr": 27.725963622212234,
"vct.col.ast.LoopStatement": 27.580825756456694,
"vct.col.ast.Dereference": 27.233676301581355,
"vct.col.ast.NameExpression": 27.13844299853828,
"vct.col.ast.OperatorExpression:Cast": 26.434398433277792,
"vct.col.ast.AssignmentStatement": 26.274336454063555,
"vct.col.ast.VectorBlock": 26.05520968526931,
"vct.col.ast.OperatorExpression:Plus": 25.81228478752888,
"vct.col.ast.OperatorExpression:LTE": 25.783569584554655,
"vct.col.ast.PrimitiveType:Float": 25.663568501170964,
"vct.col.ast.PrimitiveType:Integer": 25.623230784808186,
"vct.col.ast.PrimitiveType:Boolean": 25.557324285814573,
"vct.col.ast.OperatorExpression:And": 25.52308144955759,
"vct.col.ast.OperatorExpression:Old": 25.305706713827508,
"vct.col.ast.OperatorExpression:AbstractState": 24.676600301239876,
"vct.col.ast.OperatorExpression:GT": 24.189818628219495,
"vct.col.ast.OperatorExpression:LT": 24.128451210754214,
"vct.col.ast.OperatorExpression:Perm": 22.89239652491117,
"vct.col.ast.MethodInvokation": 22.372184138028395,
"vct.col.ast.OperatorExpression:Mult": 22.265506715207977,
"vct.col.ast.OperatorExpression:Star": 22.157972624387536,
"vct.col.ast.OperatorExpression:Tail": 22.1478887113556,
"vct.col.ast.OperatorExpression:Length": 21.487199634204508,
"vct.col.ast.Lemma": 20.77125580954531,
"vct.col.ast.OperatorExpression:Wand": 20.77125580954531,
```

```
"vct.col.ast.OperatorExpression:Div": 20.6645021339816,
"vct.col.ast.StructValue": 20.281031100252264,
"vct.col.ast.OperatorExpression:Assign": 20.083947753268713,
"vct.col.ast.PrimitiveType:Sequence": 19.946875767473053,
"vct.col.ast.OperatorExpression:Implies": 19.09527907137906,
"vct.col.ast.IfStatement": 18.99798117719227,
"vct.col.ast.OperatorExpression:Unfolding": 18.69835423706349,
"vct.col.ast.OperatorExpression:ITE": 18.563576661590258,
"vct.col.ast.OperatorExpression:NEQ": 18.012436445804454,
"vct.col.ast.OperatorExpression:NewArray": 17.885762908669058,
"vct.col.ast.OperatorExpression:FoldPlus": 17.409629425540707,
"vct.col.ast.ParallelAtomic": 16.65651487459199,
"vct.col.ast.OperatorExpression:Value": 16.234891048201902,
"vct.col.ast.OperatorExpression:Or": 12.946168512603656,
"vct.col.ast.OperatorExpression:Not": 11.43985102676475,
"vct.col.ast.OperatorExpression:HistoryPerm": 11.064281335745033,
"vct.col.ast.OperatorExpression:Member": 10.164520676691728,
"vct.col.ast.BindingExpression:Sum": 10.08634367606029,
"vct.col.ast.OperatorExpression:AddAssign": 10.08634367606029,
"vct.col.ast.OperatorExpression:Contribution": 10.08634367606029,
"vct.col.ast.OperatorExpression:ReducibleSum": 10.08634367606029,
"vct.col.ast.ParallelInvariant": 9.526194355266465,
"vct.col.ast.OperatorExpression:Future": 9.00656354663275,
"vct.col.ast.OperatorExpression:Head": 7.815945680862345,
"vct.col.ast.PrimitiveType:Process": 7.334332954372023,
"vct.col.ast.OperatorExpression:Mod": 6.632120827285922,
"vct.col.ast.OperatorExpression:History": 6.2134677553818225,
"vct.col.ast.BindingExpression:Forall:non-linear": 5.52842764546774,
"vct.col.ast.OperatorExpression:UMinus": 1.507739953132445,
"vct.col.ast.PrimitiveType:Location": 0.5229270202412382,
"vct.col.ast.OperatorExpression:Scale": 0.5229270202412382,
"vct.col.ast.PrimitiveType:Option": 0.0,
"vct.col.ast.PrimitiveType:Set": 0.0,
"vct.col.ast.PrimitiveType:Bag": 0.0,
"vct.col.ast.OperatorExpression:Append": 0.0,
"vct.col.ast.OperatorExpression:Drop": 0.0,
"vct.col.ast.OperatorExpression:NewSilver": 0.0,
"vct.col.ast.OperatorExpression:Take": 0.0,
"vct.col.ast.TypeVariable": 0.0,
"vct.col.ast.OperatorExpression:MatrixSum": 0.0,
"vct.col.ast.OperatorExpression:VectorCompare": 0.0,
"vct.col.ast.OperatorExpression:VectorRepeat": 0.0,
"vct.col.ast.OperatorExpression:OptionSome": 0.0,
"vct.col.ast.OperatorExpression:TypeOf": 0.0,
"vct.col.ast.BindingExpression:Exists": 0.0,
"vct.col.ast.OperatorExpression:Held": 0.0
```

}