Explicit sequencing of C programs

Tristan Brice Velloza Kildaire

***Abstract*—Improving software verification capabilities in Lazy-CSeq by accounting for unspecified behavior in C.**

***Index Terms*—Lazy-CSeq, C, unspecified behavior, CBMC, software verification, sequencing**

I. Introduction

THE C programming language is a widely used systems programming language used by many developers across the world through its numerous implementations. The C programming language is rather simple as well, leading to the assumption that if given a legal trivial C program such as:

```
1  #include<assert.h>
2  int counter = 1;
3  int f1() {
4          counter = counter*2;
5          return counter;
6  }
7  int f2() {
8          counter = counter*3;
9          return counter;
10 }
11 int f(int x, int y) {
12         return x+y;
13 }
14 int main() {
15         int res = f(f1(), f2());
16         assert(res==8);
17         return 0;
18 }
```

One would easily be able to deduce what the output of such a program would be - the value of `res` should be `8`, hence the assertion `assert(res==8)` should pass and compiling with `clang` affirms this. However, running this code through a compiler such as `gcc` results in an assertion failure:

```
Assertion 'res==8' failed.
```

This divergence is caused by *unspecified behavior*. Whilst a program that exhibits *undefined behavior* is an *illegal* C program, such as:

```
f(++i, ++i);
```

those who exhibit *unspecified behavior* are still legal, such as the first example. The goal of the project is to transform input code exhibiting unspecified behavior into a form that accounts for all possible legal orderings of evaluation.

A. Unspecified behavior

C leaves certain aspects of the language unspecified, meaning that the semantics for such aspects are left up to the compiler implementer to choose. Sometimes this leads to the simplest of C programs exhibiting completely different behaviors across various compilers.

What we have seen above is an example of the unspecified behavior of function argument evaluation ordering, in other words the order in which the arguments to the function `f(_, _)` are called is not specified by the C language specification hence evaluating from left-to-right (calling `f1()` followed by `f2()`) or from right-to-left (calling `f2()` followed by `f1()`) would both be legal behaviors of this program. One can start to see how the reliance on ordering of such functions which have side-effects can cause widely different answers, `res==8` for left-to-right and `res==9` for right-to-left[2]).

This is only worsened by the fact that such ordering could differ call by call and still be legal.

Such underspecified behavior also occurs within binary operations such as + whereby the first function called is not required to be `f1()` (they can be executed in any order):

```
f1()+f2();
```

B. Software verifiers

Software verification is the process of taking an input program which contains assertions or conditions that must hold true and then finding ways in which such assertions can be violated. An input program that has no assertion violations can be designated as “safe”, however a program by which such assertions are violated would be designated as “unsafe”.

Software verifiers normally make use of a technique known as symbolic execution[14] which provides a mechanism by which all paths of execution of a given input program can be explored. Whilst exploring each path these assertions are checked for if they hold or not, in the case that one assertion does not hold along any given pathway the verifier will halt and mark the program as unsafe.

Although verifiers *do* explore all different valid pathways of program execution flow they do tend to fix an ordering in the aforementioned cases of underpspecified behavior. For example, CBMC fixes an evaluation ordering of strict left-to-right when it comes to function call arguments. The result of such a fixed ordering is that CBMC will never explore the pathways that would be generated by different evaluation orderings which may result, in some cases, of assertions being violated. This leads developers of software to ship software which, despite the verification process, can still contain bugs.

A possible solution to such a problem is to design a *source-to-source* transformation which will take an input program and instrument it in such a way such that the output

program will exhibit the behavior of encompassing all valid evaluation orderings when run under a software verifier. Lazy-CSeq, which provides a source-to-source transformation framework for C programs, is an ideal candidate for implementing such a technique of which will be the focus of this paper.

II. Background

A. Sequence point

A sequence point is defined as a point within the execution of a program whereby a guarantee can be made that all the side effects of previous evaluations before the sequence point will have been completed *and* that all evaluations that follow the sequence point have had none of their side-effects performed yet[1]. The importance of sequence points within a C program are that they provide a concrete reference points whereby one can be assured that a given expression has been fully evaluated such that the resulting value of it can be tested with confidence.

An expression can be *sequenced-before* that of another expression, where the latter is then *sequenced-after* the former - hence an explicit ordering is present. An expression can be *unsequenced* meaning that there can be an overlap in execution. The types of sequence points we are interested in are what are referred to as *indeterminately sequenced*. These sequence points refer to a case where the evaluation of expressions A and B must fully complete but the order in which A and B are evaluated is unspecified.

B. CBMC

CBMC (*C Bounded Model Checker*) is a software verifier which allows one to provide C source files for verification to ensure that certain guarantees set out by the programmer are met. At its core level it will explore all possible pathways of execution of a given C program and for each of these ensure that the `assert()`s added by the programmer pass (evaluated to `true`). Below an example is provided which shows 3 pathways that can be explored:

```
1  int x = nondet_int();
2
3  if(x == 0) {
4    assert(1);
5  }
6  else if(x == 5) {
7    assert(1);
8  }
9  else {
10   assert(0);
11 }
12
13 __CPROVER_assume(x == 0 || x == 5);
```

This example[3] contains a local variable `x` which is assigned `nondet_int()`, this is a CBMC subroutine which returns a non-deterministic value. Meaning that `x` hold any value possible for a 32-bit integer and CBMC will explore all pathways for each of those cases. However, in this example we add a call to `__CPROVER_assume()` and provide it a condition, `x == 0 || x == 5`, this forces the symbolic execution to ensure that when the assumption statement is reached that said condition is true. The effect this has it the culling of the search space to a case whereby `x` will ever only be $x\epsilon\{0, 5\}$, therefore the only branches explored will be the first two (lines 3 and 6), which happen to be where the assertions pass, as compared to the last case where it will fail otherwise if not for this bounding.

This provides a software developer writing unit tests that run under CBMC some control as to how the exploration of pathways is allowed to grow. An example of this would be take both paths of execution in a branch statement, such that a program like this.

C. Lazy-CSeq

Lazy-CSeq is a tool which performs the explicit sequencing of C programs. It contains several source-to-source transformations that are applied in succession (i.e. the output of one transformation is the input of another). resulting in a final transformed source file. These transforms have the goal of explicitly sequencing several aspects of the C constructs present in source code which entails breaking down complex statement and expressions into simpler, more atomic, statements. An explicitly sequenced version of a C program entails higher level of granularity between statements where Lazy-CSeq can insert context switching code in between, as to simulate as best as possible real life context switching but on a sequential form of the original program. This is done with the goal to allow realistic multi-threaded program analysis[10].

One of the various explicit sequencing transformations is that the in-lining of function calls and their nested arguments, such that our original program can be written as this:

```
1 int __f1_value = f1();
2 int __f2_value = f2();
3 f(__f1_value, __f2_value);
```

Now instead of a single line, a context switch can be inserted between the calls to `f(_,_)`'s arguments. It is within this stage of the Lazy-CSeq pipeline that we want to implement our transformation that can account for the orderings of evaluations of `f1()` and `f2()` in this example. We are required to do this transformation as Lazy-CSeq currently enforces strict left-to-right evaluation ordering of arguments.

Another transformation we are interested in is that of binary operations discussed earlier:

```
1 int __f1_value = f1();
2 int __f2_value = f2();
3 __f1_value+__f2_value;
```

It is here *too* where we shall handle binary operations and their various orderings of evaluation.

III. State of practice

Keeping in mind that we want to develop a mechanism by which *all* possible permutations of ordering would be

accounted for, it would be instructive to look into which subset, if any, of the orderings are actually implemented in practice by the compilers of today. Therefore a testing script was developed by which programs like that of the introductory piece could be tested for assertion passes/failures on various compilers, standard flags and optimization levels.

We tested the following compilers and software verifiers:

1) `clang`, `tcc`, `ccomp`, `gcc`, `cbmc`, `esbmc`

Per each of the above, the following C standards were tested (with the exception that `ccomp`, `cbmc` and `esbmc` do not support specifying these):

1) `c99`, `c9x`, `c90`,
2) `c2x`, `c17`, `c18`, `c11`, `c1x`
3) `gnu11`, `gnu89`, `gnu90`, `gnu99`,
4) `iso9899:1990`,`iso9899:199409`, `iso9899:1999`, `iso9899:2011`, `iso9899:2017`

The point in testing the standard flags was to see if there was any case whereby this could change behavior of a given compiler.

Per each of the above, the following optimization levels were tested (with the exception that `cbmc` and `esbmc` do not support specifying these): `-O0`, `-O1`, `-O2`, `-O3`

Similarly we wanted to see if the different levels of code optimization could cause differing behavior within a given compiler.

Two main aspects of the C programming language whereby unspecified behavior is present are in the aforementioned function call argument evaluation ordering and list-initializer element evaluation ordering.

A. Function call argument evaluation ordering

The tests[6] showed that the following compilers `cbmc`, `esbmc`, `clang`, `ccomp`, `tcc` prefer left-to-right ordering:

`gcc` preferred right-to-left ordering.

B. Binary operation ordering

The tests[7] showed that `cbmc`, `esbmc`, `clang`, `ccomp`, `tcc`, `gcc` prefer left-to-right ordering.

C. List-initializer element evaluation ordering

The order in which the elements of a list initializer[8] are evaluated is unspecified:

```
int results[2] = {f1(), f2()};
```

However, from our findings no compiler or verifier differed to one another, all performed left-to-right evaluation ordering in this case.

Although in practice both list-initializers and binary operations show no ordering differences we have chosen to include the latter in our transformation procedure due to the fact that binary operations may occur in nested expressions whilst list-initializers may not. For this reason we have also decided to implement handling for them at a statement-level as an extra feature.

IV. Solution

We need to effectively design a transformation which can allow for CBMC to see all the $n!$ (where n is the number of arguments to the function call/binary operation) argument evaluation orderings when exploring control flow pathways. In reality only two permutations of orderings would be required (strict left-to-right and strict right-to-left) but it is much easier implementing a general form of this for all permutations and would also future proof the design if any compiler decides to choose a more exotic ordering in the future. Here we discuss the code transformation itself and how it accomplishes this and then the meta-programming required to generate the code transform.

A. Overview

The transformation takes in an expression exp which exhibits *unspecified sequencing*, splits up the expression into further sub-expressions - recursively re-applying this procedure till the sub-expression no longer exhibits under-specified sequencing. The result is n-many sub-expressions $e_0 ... e_n$ which do not exhibit *unspecified sequencing*. A loop over these n-many sub-expressions is then generated whereby each sub-expression e_i gets a chance to be evaluated in an unspecified order but with the guarantee that each e_i will be evaluated and at most once. Built-in dependency management assures that any expression e_i which requires sub-expressions e_k, e_j (with $k \neq j \neq i$) to be evaluated *before* itself is enforced - therefore ensuring semantic correctness in those cases.

This mechanism provides us with at most $n!$ orderings possible for evaluation ordering ensuring we account for them all and in a valid manner.

B. Transformation

Each argument i to a function call and each operand to a binary operation respectively gets its own pair of variables (e_i, s_i) which are the *evaluation variables* and *status variables* respectively. The evaluation variable holds the evaluated value of the argument/expression and the status variable indicates whether or not a given argument is still a candidate for evaluation. Initially we have $s_i = 1 \forall i$ - indicating that each argument is candidate for evaluation. Lastly, we denote the total number of arguments and operands needing to be evaluated as n.

A loop over the range $[0, n)$ is generated which several items, pre-iteration code, branches and post-iteration code.

1) Transforming $f(f1(), f2())$ (simple example): For a simple starting example we look at the case of transforming the function call`f(f1(), f2())`, below we have included the full transformed code, followed by an explanation.

```
1  int e_0 = 0, e_1 = 0, e_3 = 0;
2  int s_0 = 1. s_1 = 1, s_2 = 1;
3  int n = 3;
4
5  for(int i_0 = 0; i_0 < n; i_0++)
6  {
7    magic_0 = nondet_uint();
8
9    if(magic_0 == 0 && s_0 && 1) {
10     e_0 = f1();
11     s_0 = 0;
12   }
13   if(magic_0 == 1 && s_1 && 1) {
14     e_1 = f2();
15     s_1 = 0;
16   }
17   if(magic_0 == 2 && s_2 &&
18     (s_0 == 0 && s_1 == 0)) {
19     e_0 = f(e_0, e_1);
20     s_3 = 0;
21   }
22 }
23
24 __CPROVER_assume(s_0 == 0 &&
25                  s_1 == 0 &&
26                  s_2 == 0);
```

Lines 1 to 3 define our (e_i, s_i) pairings which are the *evaluation variables* and *status variables* respectively. We have it such that $i\epsilon[0, n)$ with $n = 3$ (line 3) in this case as there are three expressions requiring evaluation `f1()`, `f2()` and `f(_, _)`.

From line 7 on wards we iterate n-times with each iteration beginning with a sourcing of non-determinism by calling `nondet_uint()` and storing this result into `magic_0`. Following this we now have 3 sets of branches that can run on lines 9, 13 and 17.

The value of `magic_0` is what will determine which of the 3 branches run in any given iteration, giving equal chance per-iteration for each expression to be evaluated, the value is bounded by $magic\epsilon[0, n)$. Any given branch will only be executed if it has not already been run before, in other words a branch i is candidate for execution when $s_i = 1$ *and* if its dependent branches have all been run, $s_j = 0 \forall j \epsilon \{dep1, dep2\}$.

Each branch i contains two simple components, the evaluation of the expression e_i and the setting of the status variable s_i. A function call such as that `f1()` will first be inlined rather than an actual function call, the resulting evaluation of the inlined expression is then stored in its evaluation variable $e_{i=0}$. Following this the branch is marked as completed with $s_i = 0$ ensuring it is no longer candidate for execution in the following iteration *and* that any other branches dependent on it (using $e_{i=0}$ as part of a sub-expression) can safely assume that the evaluated expression is available for usage.

An example of a branch with dependencies is that of the final evaluation of `f(_, _)` (line 19) as it requires both `f1()` (the first branch - line 9) and `f2()` (the second branch - line 13) to have already been evaluated ($s_0 = 0 \,\&\&\, s_1 = 0$) as it makes use of e_0 and e_1 in its expression $e_2 = f(e_0, e_1)$. Therefore only the last branch will always be executed last however the first two have equal probability of being run on the first iteration.

The last line, 24, contains a call to `__CPROVER_assume(s_0 == 0 && s_1 == 0 && s_2 == 0)` ensures that CBMC only explores all the pathways of execution that meet the condition of $s_i = 0 \forall i \epsilon [0, n)$, essentially implying that all branches must, in which ever order, *somehow* all be run (at most once). This has the effect of ensuring we obtain all possible *valid* evaluation orderings possible. This is also what bounds the value of $magic$ to $[0, n)$ (as mentioned earlier) which would have otherwise been any valid 32 integral value as defined by the `int` data type.

2) Transforming $posReturn() \,\&\&\, f(count1(), count2())$ (conditional example): In the case whereby a full expression contains *only* one conditional binary operation such as the logical AND (&&), we are therefore required to always execute the left-hand operand, $posReturn()$, but then only execute the right-hand operand, $f(count1(), count2())$, if the left-hand operand had a non-zero evaluation ($posReturn() \neq 0$). Therefore we have to add special conditional handling in our code transformation for both the positive ($posReturn() \neq 0$) and negative case ($posReturn() = 0$).

```
1  int e_0 = 0, e_1 = 0, e_2 = 0, e_3 = 0, e_4 = 0;
2  int s_0 = 1, s_1 = 1, s_2 = 1, s_3 = 1, s_4 = 1;
3  int n = 4;
4
5  for(int i_0 = 0; i_0 < n; i_0++)
6  {
7    magic_0 = nondet_uint();
8
9    if(magic_0 == 0 && s_0 && 1) {
10     e_0 = posReturn();
11     s_0 = 0;
12   }
13   if(e_0) {
14     if(magic_0 == 1 && s_1 && 1) {
15       e_1 = count1();
16       s_1 = 0;
17     }
18     if(magic_0 == 2 && s_2 && 1) {
19       e_2 = count2();
20       s_2 = 0;
21     }
22     if(magic_0 == 3 && s_3
23                     && (s_1 == 0
24                     && s_2 == 0)) {
25       e_3 = f(e_1, e_2);
26       s_3 = 0;
27     }
28   }
29   else {
30     s_1 = 0;
31     s_2 = 0;
32     s_3 = 0;
33   }
34   if(magic_0 == 4 && s_4
```

```
35                      && (s_1 == 0
36                      && s_2 == 0
37                      && s_3 == 0
38                      && s_0 == 0)) {
39        e_4 = e_0 && e_3;
40        s_4 = 0;
41      }
42    }
43
44    __CPROVER_assume(s_0 == 0 &&
45                     s_1 == 0 &&
46                     s_2 == 0 &&
47                     s_4 == 0);
```

The transformation is almost identical to the previous example but with the additional conditional branch `if(e_0)` (line 13) which contains code to run the sub-expressions `f(count1(), count2()` when $e_0 \neq 0$, in the case $e_0 = 0$ then we must not run that code (as per the C language rules), however we then run the `else` (line 29) branch which ensures that we mark those un-run expressions as "completed" (to satisfy the CBMC assumption that follows - line 44) by ensuring $s_1 = s_2 = s_3 = 0$ (lines 30 to 32).

Following this the final computation of the binary operation is computed with `e_4 = e_0 && e_3` (line 39).

C. Implementation

1) Overview: Lazy-CSeq is written in Python2 and is a relatively simple program. Lazy-CSeq is composed of a set of Python modules that reside in a directory named `modules/` along with so called *"chain files"*. A *chain* is simply a plain text file containing a list of modules to be run on the given C source code file. The order with which the modules are listed is significant as it is the order in which the transformations will be applied.

2) Parsing with `PyCParser`: Lazy-CSeq uses a pre-existing C parser called PyCParser[9] which is used to read the source files in and construct an AST (Abstract Syntax Tree) tree of all the nodes in the source file such that they can be easily manipulated in Python. Each module contains a class which is a sub-type of `CParser` effectively making each Lazy-CSeq module a kind-of parser. It is with this mechanism that each module implements specific behavior when encountering certain types of AST nodes by method of overriding specific visitation functions that are called when traversing a certain type of syntactical component. For example `visit_FuncCall()` is the method called whenever a function call such as `f1()` is encountered in any expression, similarly whenever a binary operation such as `f1()+f2()` is encountered then `visit_BinaryOp()` is called. These methods are passed a node object representing the AST node, transformations and further nested visitation function calls are made and the result is a string of the transformed input expression. It is mainly within these methods whereby the implementation of our code transformation module resides.

3) Chained transformation: The process of sequencing in Lazy-CSeq works by iterating over the chain file, instantiating a given module by providing the input source code as a string, running the transformations and then returning the transformed result as a string. This new resultant string will then be used as the input for the next module in the chain and so on. Each module performs a specific type of transformation that is then processed further by another module that performs some other sort of transformation.

The specific modules are `t_preinliner.py` and `t_unroller.py` which handle the pre-processing for inlining function calls and the unrolling of the generated for-loops.

Our implementation entails the creation of a new chain file based off of that of the current standard chain named `lazy.chain` but instead of using their function pre-inliner (`preinliner.py`) and general unroller (`unroller.py`) we use a new ones developed for the sole purpose of transforming function calls and binary operations in the manner described earlier.

The new chain can be activated in Lazy-CSeq by specifying the `-l` flag and providing the path to the chain file, like:

```
$ ./cseq.py -l lazy_tristan2 -i inputfile
```

Where`inputfile` is the C source file to process, optionally the `-D` flag can be passed as well to show informative debugging messages.

D. Implementation details

`t_preinliner` takes care of transforming expressions containing function calls and binary operations. Whenever one of these two are encountered at the statement level a so-called "sanity check" is run to see if the transformation can be properly applied in this case. If the transformation can be applied then we go ahead, however, if the check fails then the transformation is not applied and we move onto the next statement - leaving the statement entirely un-transformed.

These class of expressions are visited recursively till the leaf nodes are met, before each recursive visitation is invoked a new "Entry" is created and pushed onto a stack. This is then used by the next call as a way to attach further entries to their parent entries - creating what is in affect a parse tree.

Once the recursion unrolls we are left with a top-level Entry which is then passed to the transformation function. This function flattens the tree in a linearized fashion, items in this list of then processed and dependencies between nodes are built - this is how branches are generated one by one and how branches with dependent status variables are resolved. Each node in this list is handled slightly differently but all make use of the evaluation stack which holds the inlined variable names such that these are placed in the correct expressions when needs be - hence quasi-recursive nature is maintained in order to process a linear list as though it were a tree of sorts representing the program's structure.

Following this, a preamble is generated which declares the status variables, evaluation variables, magic number and

loop facilities. In between these and the boundary assumption the branches generated are then placed.

1) Post-transformation fixups: Due to the fact that the the for-loop unrolling process which occurs before the generation of our for-loops and due to the necessity that our generated for-loops require unrolling as well we need to call the unrolling process again after our transformation. The `unroller.py` inserts labels named in an incremental manner based on an internal counter initialized during module start-up, since we will instantiate a similar unrolling module for *our* for-loops via `t_unroller.py` we will have this counter initialized to zero - causing duplicate labels to be produced. `t_unroller.py` both provides unrolling for the transformation-generated for-loops along with ensuring that duplicate labels to not appear in the final transformation.

V. Results of benchmarking

The benchmarking process involved two types of tests:

1) Sequential
   a) These test cases were done to simply test the transformation's raw performance impact on a single-threaded program
2) Concurrent
   a) These test cases involved programs with multiple threads (via the `pthread` library) in order to see the affect the transformation would have alongside the explicit sequencing transformation on such programs

The measurements conducted are all related to the run times of the following three sub-systems in the CBMC pipeline:

1) Solver times
   a) When CBMC analyzes a program it generates a system of equations which represent the path conditions, such as those imposed by `assert()` statements, along the several pathways explored. In order to determine whether or not a condition holds throughout all the pathways this system of equations needs to be solved - which can be a time consuming process.
2) Symbolic execution times
   a) CBMC creates a symbolic variable for every real variable present in the input program. These hold no concrete value but rather condition based on what would be valid on a certain pathway of execution. The generation, tracking, updating and validation of these variables is too a time consuming process.
3) Decision times
   a) Certain states will contain conditions imposed by the source program itself, such as branches. CBMC must then take all the pathways possible from this point onward. The decision to determine which of these branches are feasible requires evaluating the path condition combined with several varying conditions per pathway. The further along the program - the more intense the process.

The original Lazy-CSeq is labeled as `vanilla` whilst our additions are labeled `tristan`.

A. Sequential programs

This section shows the benchmarking results on sequential programs from `benchmarks/sequential/`, these programs lack any pthreading in them and are here to show raw performance differences on singularly-threaded C programs undergoing Lazy-CSeq's transformations.

1) Solver times:

	test	vanilla	tristan
0	binary_op_bit_neg.c	0.0000s	0.0000s
1	binary_op_bit_pos.c	0.0000s	0.0250s
2	constants.c	0.0000s	0.1735s
3	for_loops.c	0.0000s	0.0446s
4	global_id.c	0.0000s	0.1712s
5	local_id.c	0.0000s	0.1892s
6	nested_calls.c	0.0002s	1.1891s
7	types_tester.c	0.0000s	0.0792s

2) Symbolic execution times:

	test	vanilla	tristan
0	binary_op_bit_neg.c	0.0032s	0.0084s
1	binary_op_bit_pos.c	0.0029s	0.0608s
2	constants.c	0.0032s	0.1082s
3	for_loops.c	0.0031s	0.0730s
4	global_id.c	0.0031s	0.1196s
5	local_id.c	0.0040s	0.1969s
6	nested_calls.c	0.0096s	1.6433s
7	types_tester.c	0.0030s	0.0727s

3) Decision times:

	test	vanilla	tristan
0	binary_op_bit_neg.c	0.0000s	0.0000s
1	binary_op_bit_pos.c	0.0004s	0.0451s
2	constants.c	0.0004s	0.2340s
3	for_loops.c	0.0000s	0.0710s
4	global_id.c	0.0005s	0.2313s
5	local_id.c	0.0006s	0.2689s
6	nested_calls.c	0.0013s	1.8964s
7	types_tester.c	0.0004s	0.1155s

B. Summary

1) Solver times summary:

	Solver	mean	min	max	median
0	vanilla	0.0000s	0.0000s	0.0002s	0.0000s
1	tristan	0.2340s	0.0000s	1.1891s	0.1252s

2) Symbolic execution times summary:

	Symex	mean	min	max	median
0	vanilla	0.0040s	0.0029s	0.0096s	0.0031s
1	tristan	0.2854s	0.0084s	1.6433s	0.0906s

3) Decision times summary:

	Decision	mean	min	max	median
0	vanilla	0.0005s	0.0000s	0.0013s	0.0004s
1	tristan	0.3578s	0.0000s	1.8964s	0.1734s

C. Concurrent programs

We now look at the performance of the transformation on concurrent benchmarks from `benchmarks/concurrent/` which contains pthreading in them, these contain a lot more instrumentation to support the simulation of context switches and therefore have more of a performance penalty. These were sourced from `pthread`[12] and `pthread-deagle`[13] from `SV-benchmarks`.

1) Solver times:

	test	vanilla	tristan
0	bigshot_p.c	0.0016s	0.0022s
1	bigshot_s.c	0.0011s	0.0012s
2	fib_safe-5.c	5.8098s	195.2360s
3	fib_safe-6.c	57.9977s	894.0980s
4	fib_unsafe-5.c	2.4312s	72.7475s
5	fib_unsafe-6.c	22.7900s	311.9150s
6	peterson.c	0.0008s	0.0008s
7	queue.c	1.0188s	1.2020s
8	triangular-1.c	0.0000s	0.0000s
9	twostage_3.c	0.0000s	0.0000s
10	safestack_relacy.c	0.2621s	0.8571s
11	airline-5.c	1.3330s	2.2133s
12	reorder_c11_bad-10.c	6.9113s	6.9701s
13	reorder_c11_good-10.c	0.0000s	0.0000s

2) Symbolic execution times:

	test	vanilla	tristan
0	bigshot_p.c	0.0107s	0.0106s
1	bigshot_s.c	0.0116s	0.0104s
2	fib_safe-5.c	0.0931s	42.5815s
3	fib_safe-6.c	0.1168s	58.9096s
4	fib_unsafe-5.c	0.1090s	37.3924s
5	fib_unsafe-6.c	0.1270s	68.4483s
6	peterson.c	0.0115s	0.0117s
7	queue.c	0.6713s	0.9351s
8	triangular-1.c	0.0112s	0.0418s
9	twostage_3.c	0.0386s	0.0480s
10	safestack_relacy.c	0.1859s	0.5312s
11	airline-5.c	0.1614s	0.2743s
12	reorder_c11_bad-10.c	0.7960s	0.8814s
13	reorder_c11_good-10.c	1.0561s	1.2726s

3) Decision times:

	test	vanilla	tristan
0	bigshot_p.c	0.0040s	0.0044s
1	bigshot_s.c	0.0036s	0.0037s
2	fib_safe-5.c	5.8627s	195.7050s
3	fib_safe-6.c	58.0683s	894.7300s
4	fib_unsafe-5.c	2.4806s	73.1691s
5	fib_unsafe-6.c	22.8596s	312.5610s
6	peterson.c	0.0025s	0.0028s
7	queue.c	1.2426s	1.4640s
8	triangular-1.c	0.0022s	0.0118s
9	twostage_3.c	0.0096s	0.0111s
10	safestack_relacy.c	0.3657s	1.1336s
11	airline-5.c	1.4751s	2.4076s
12	reorder_c11_bad-10.c	7.9253s	8.0105s
13	reorder_c11_good-10.c	0.0000s	0.0000s

D. Summary

1) Solver times summary:

	Solver	mean	min	max	median
0	vanilla	7.0398s	0.0000s	57.9977s	0.6405s
1	tristan	106.0888s	0.0000s	894.0980s	1.0295s

2) Symbolic execution times summary:

	Symex	mean	min	max	median
0	vanilla	0.2429s	0.0107s	1.0561s	0.1129s
1	tristan	15.0963s	0.0104s	68.4483s	0.7063s

3) Decision times summary:

	Decision	mean	min	max	median
0	vanilla	7.1644s	0.0000s	58.0683s	0.8042s
1	tristan	106.3725s	0.0000s	894.7300s	1.2988s

E. Analysis

There is a base overhead that is imposed on all programs tested, both sequential or concurrent, under the condition that they contain at least one statement which is candidate for transformation. Such statements are those that exhibit underspecified behavior of which all the benchmarks tested do contain. The significance of such an overhead is that the code transformation added to each test case is a costly one. Running the benchmarks *without* the transformation means that no additional for-loops will appear in the final program - however running the benchmarks *with the transformation* means that for every n-many statements in the program that are candidates for transformation, we then have an additional n-many for loops added to the program with an iteration count dependent on each n_i's argument count (a_i).

Each of these for-loops has a_i-many tuples of the variables (s_i, e_i) associated with them, resulting in a linear (with respect to the size of the AST) increase in the *total* amount of variables in the generated program. Along with this a_i-many if-statements are generated and to be run a_i-many times.

The end product of such a transformation is rather expensive.

CBMC will take all the variables of a given program input into it and assign a symbolic variable to each. Every time a symbolic variable is used within an if-statement CBMC will have to split pathways to explore, therefore duplicating the search space. Because the our transformation *uses* these variables we have created in if-statements it is therefore a guarantee this will happen. This process of splitting pathways is a normal part of symbolic execution but because we are now applying this to non-user code we are adding a significant overhead to the pathway exploration that would occur in a non-transformed program. Each iteration of a loop n_i will cause such an exploration to happen, with a_i-many iterations and a reasonable number of n_i for loops the search domain explodes. This *is* however what one would expect when attempting to take into account various different evaluation orderings.

Where we see a big explosion in run time, more so than the average, is in user-provided code which already contains for-loops which within themselves contain statements that are candidate for transformation. This results in the transformation code, which as explained is already expensive, being repeated many more times over. Such is the case within the Fibonacci examples - these contain for-loops with bodies that are candidate for transformation - this explaining their rather high run times for CBMC. The time increase in all the other programs (where this is not the case) is noticeable but not explosive.

VI. Conclusion

The C programming language exhibits under-specified behavior in certain aspects of its syntax which leads to the implementer of the respective compiler to choose a fixed but arbitrary evaluation ordering for syntactical components such as function call arguments, binary operations and list-initializer elements. Through the researching phase of this project the only notable differences between implementations was seen within function call arguments whilst binary operations and list-initializer evaluation ordering was consistent across implementations. A solution to encode the various orderings was therefore needed for Lazy-CSeq and the chosen syntactical components to be transformed was function call arguments (as there was variation) and binary operations (as an extra feature seeing as all implementations were consistently left-to-right). List-initializers were in another domain of themselves implementation-wise and therefore was not considered as part of the encoding scheme.

The solution implemented was in the form of a code transformation which would take a C program as *input* and then produce a code block as *output*. This code block contained a preamble to setup variables to hold evaluations of expressions, a for-loop which provided a mechanism of evaluating function call arguments and binary operation operands in a manner that each would have a turn to be evaluated non-deterministically at any given iteration. This *output* code would be fed into CBMC which would handle the exploration of the various pathways that are represented by such non-deterministic code, therefore exploring the various different evaluation orderings for us.

The results from the benchmarking process show a significant increase in run times in both sequential (non-concurrent) and concurrent programs. The mean symbolic execution time has increased due to the significant increase in the number of symbolic variables that are present in the transformation code. Along with this is a noticeable increase in the decision time due to the number of additional branch statements (if-statements) that is contained in the new transformation. The increased number of symbolic variables and conditions (required by the branches) within the system of equations that CBMC produces means that the solving of such a system has an increased run-time, leading to the solver times having a higher average run-time.

A general improvement in run time was made by removing Lazy-CSeq's hard requirement of using CBMC `5.4.0`. Using the newest version, as of writing `5.50.0`, resulted in a slight performance increase in the time taken to verify the transformed code generated by Lazy-CSeq.

Some of the shortcomings of the parser-based approach that Lazy-CSeq takes is that the visitation strategy of, for example, binary operations like `f1()+f2()+f3()` is inherently left-to-right in the sense that `f1()+f2()` will be processed in a group first and *then only* will `res+f3()` be processed, therefore we cannot do a level-traversal of all operands in an equal manner. A future improvement would be to fix this however it would require a significant change from the parser's current left-to-right depth-first search strategy.

As described earlier list-initializers exhibit unspecified behavior however in practice no compiler of the day deviates from left-to-right evaluation ordering. However, to be safe - such a feature could be added to Lazy-CSeq to catch all orderings that *may* be in practice in the future. Adding support for this would effectively future-proof Lazy-CSeq with respect to under specified behavior.

In conclusion, this feature addition to Lazy-CSeq provides the tool-chain with a more holistic software verification mechanism which improves upon the previously hard-coded left-to-right evaluation ordering strategy, such that we can explore the previously unexplored territory of under-specified behavior which until now may have hidden potential bugs from software developers testing software which exhibited such behavior.

VII. Appendix

A. `lazy_tristan2` chain file

The new chain appears as follows:

```
## Program simplification
workarounds
functiontracker

preinstrumenter
constants
spinlock

## Loop and control-flow transformation
switchtransformer
dowhileconverter
conditionextractor

## Program flattening
varnames
functionpointer

## Program flattening
## (function call handling, preinilining)
unroller
t_preinliner

## Program fattening
## (remaining inlining, etc.)
inliner

## Program flattening
## (Unrolling of transformation loops)
t_unroller

## Sequentialization
duplicator
condwaitconverter
lazyseq

## Instrumentation
instrumenter

## Analysis
mapper
feeder
cex
```

References

[1] *Sequence point*, https://en.wikipedia.org/wiki/Sequence_point

[2] *Order of evaluation*, https://en.cppreference.com/w/c/language/eval_order

[3] M. Frade, *CBMC by example*, https://haslab.github.io/MFES/2122/CBMCexamples-handout.pdf

[4] T. Velloza Kildaire, *CBT Testing framework*, https://github.com/cbtorture/tests.

[5] T. Velloza Kildaire, *Clang Behaviour Table*, https://cbtorture.github.io/findings/.

[6] T. Velloza Kildaire, *Function Call Evaluation ordering*, https://cbtorture.github.io/findings/f1/

[7] T. Velloza Kildaire, *Binary operator test*, https://cbtorture.github.io/findings/binop/

[8] T. Velloza Kildaire, *List initializer element evaluation ordering* , https://cbtorture.github.io/findings/l1/

[9] E. Bendersky, *Complete C99 parser in pure Python*, https://github.com/eliben/pycparser

[10] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, *Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization*

[11] T. Velloza Kildaire, *Level-order test for function evaluation*, https://cbtorture.github.io/findings/f69/

[12] , *Various small concurrent programs.*, https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c/pthread

[13] , *Some concurrent programs writing in C with Pthread.*, https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c/pthread-deagle

[14] , *Symbolic execution*, https://en.wikipedia.org/wiki/Symbolic_execution