QUERY OPTIMIZATION FOR SYMBOLIC EXECUTION

It is well-known that, while doing symbolic execution, the running time is dominated by SMT solvers checking satisfiability of these path constraints. Quite a few different techniques have been used over the years to reduce this time. In the same vein, I have studied three new optimizations in this paper. All of them are then evaluated on a few sample programs.

Arsalan Ali Gohar Jumani
2017-03-0010

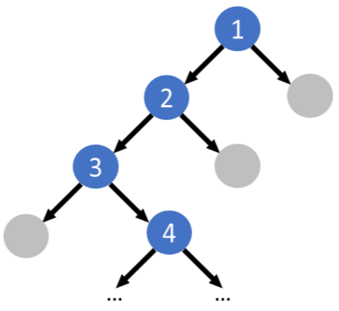
Name: Arsalan Ali Gohar Jumani *Final Project Report*
Roll #: 2017-03-0010

Contents

Name: Arsalan Ali Gohar Jumani
Roll #: 2017-03-0010

Introduction

When a program is symbolically executed, each forking statement leads to two child states that need to be further analysed symbolically. This ensures that the nature of path constraints is incremental, i.e. all predicates except one are shared between a child state and its parent. The tree diagram on the right illustrates the situation. The table below shows the statement associated with each forking state and how that leads to incremental path constraints.



Node	Statement	Path Constraint
1	S_1	–
2	S_2	S_1
3	S_3	$S_1 \wedge S_2$
4	S_4	$S_1 \wedge S_2 \wedge \neg S_3$

It is well-known that, while doing symbolic execution, the running time is dominated by SMT solvers checking satisfiability of these path constraints. Quite a few different techniques have been used over the years to reduce this time. In the same vein, I have studied three new optimizations in this paper. All of them are then evaluated on a few sample programs.

Literature Review

Some of the optimizations found in the literature have been categorized below [1]. Note that all of these techniques led to a significant reduction in the time spent solving constraints. Even then constraint solving remains a bottleneck as SMT queries generated by symbolic execution are NP-complete.[1]

Constraint Set Reduction

It concerns simplification of the path constraints or queries. Some of the optimizations include finding independent subsets / subqueries for a given path constraint and substituting out a symbolic value once it is concretized [2].

Reuse of Constraint Solutions

It involves techniques as obvious as caching of queries already solved [3] to more sophisticated ones using complex data structures to search for, say, satisfiable query supersets [2]. It also involves techniques which expresses constraint sets, in canonical forms [4], as logical implication relations [5] for better reuse across runs of the same or similar programs, different programs or different analyses.

Lazy Constraints

It involves solving constraints lazily when expensive operations are involved in the path constraints [6].

Optimizations

This section explains the optimizations that were tested using Z3 [7], an SMT solver developed by Microsoft Research. The tests involved solving path constraints of some C++ functions, as given in the Appendix. A Ubuntu 16.04 LTS virtual machine running on Lenovo™ ideapad™ 510S (8GB RAM, Intel® Core™ i7-7500U CPU @ 2.7GHz 2.9GHz) was used to carry out the tests in 'best performance' mode.

[1] Quantifier-free formulas in theory of bit-vectors and theory of arrays are decidable.

Retaining Learned Clauses for Incremental Solving

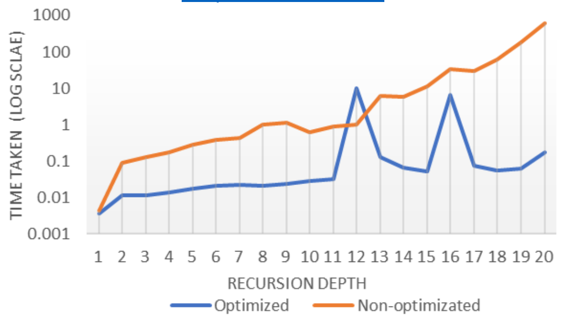
The efficiency in solving SMT queries is driven by learning of clauses[2] and unit propagation [8]. This ensures the subtrees where no solution was found are trimmed going forward. The learned clauses are implied by the clauses in the query. Hence, any superset of the query will also imply them.

Given the incremental nature of the path constraints, solving efficiency will be improved if a child state inherits the learned clauses from its parent instead of deriving them from scratch[3].

Results

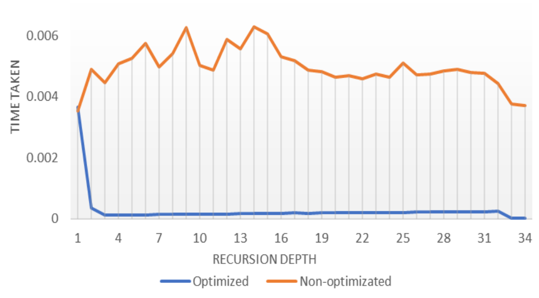
It is clear that retaining learned clauses in the child state whenever a state forks offers quite a significant gain in efficiency, sometimes by many orders of magnitude.[4]

Simple GCD Function



This function was solved till recursion depth of 20 once. After the 20th iteration, it was not feasible to solve the non-optimized queries.

LOG Function



This function was solved till recursion depth of 34 ten times. The depth of 33-34 represent "unsatisfiable" queries.

Choosing Between Deep Copy of Solver Object and Forking the Process

When a state forks, there are two possible ways to proceed with the symbolic execution of the resultant child states:

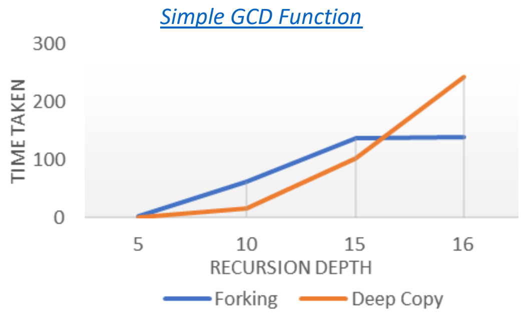
i. **Forking:** execute them both in separate threads.[5]
ii. **Deep Copy:** make a deep copy of the parent solver object and execute the child states in the same thread.

Both of these possibilities are trying to leverage the efficiency gained by retaining parent state's learned clauses. So, I compared which one performs better. Intuitively, a solver object interacting heavily with the environment might invoke many expensive copy(-on-write) operations.

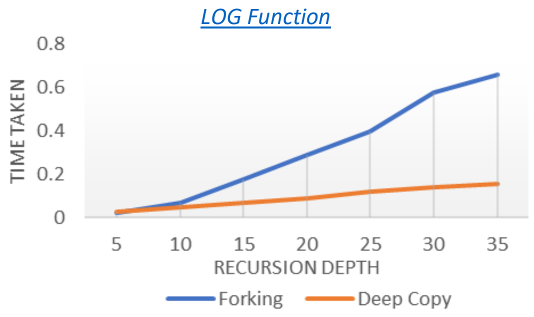
Results

While the graphs show mixed results, the results are actually incomparable. The way Z3 heuristically decides which portions of search space to give priority gets affected by the total number of constraints in a thread, be they spread out in different solver objects. Other solvers do not offer an API call to make an independent copy of a solver or make it difficult to enforce determinism in search for solution.

[2] Generated by both the theory solver and underlying SAT solver.
[3] Most theory solvers, including Z3, will not generate the same set of learned clauses in the child state. But the overlap will be significant.
[4] All symbolic execution engines I have come across do take advantage of this.
[5] It is essential to use this possibility in some form and to some degree to efficiently use a multiprocessing environment.



This function was solved till recursion depth of 16 once. From 17th iteration onwards, it was not feasible to solve the deep copy.



This function was solved till recursion depth of 35 ten times. The depth of 33-35 represent "unsatisfiable" queries.

Utilizing Partial Cached Solutions

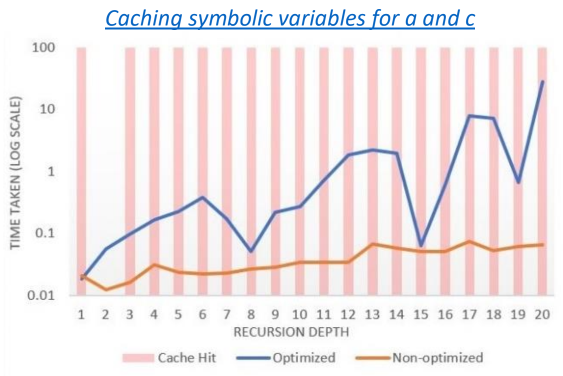
'Counterexample caching' in [2] was used to quickly check if solution to a previously solved query also satisfies its superset that we have obtained as a path constraint. If not, we run the solver on the path constraint. One way to extend this optimization is to enforce a partial assignment in the solver object obtained from a satisfiable subset. If partial assignment could also form part of the solution to the original query, intuitively it should enhance efficiency greatly due to the greatly restricted search space.

The modified version of the GCD function was used to test this. The incremental path constraints were solved using the first optimization as a benchmark for comparison. The optimized version worked as follows:

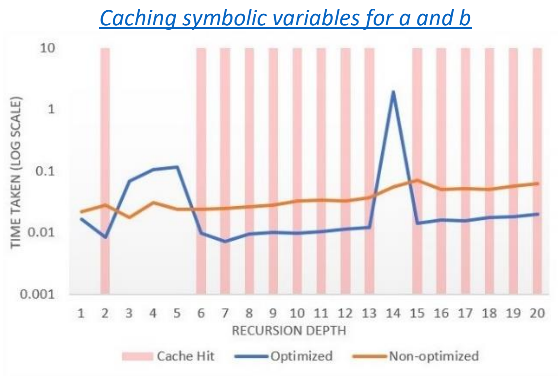
i. Use solution from the last query solved using the non-optimized version partially.
ii. Enforce the obtained partial solution in the solver object for the current path constraint.
iii. Run the modified solver object. If it's unsuccessful, we run the solver again just like it is run in the benchmark case.

Results

The results were mixed, depending upon which subset of variables were cached. The cache hits on the other hand were extremely frequent. This suggests that the solver is not optimized to take in partial solutions. Moreover, using solutions like 'implied value concretization' from [2] when enforcing the partial solution will lead to loss of learned clauses.

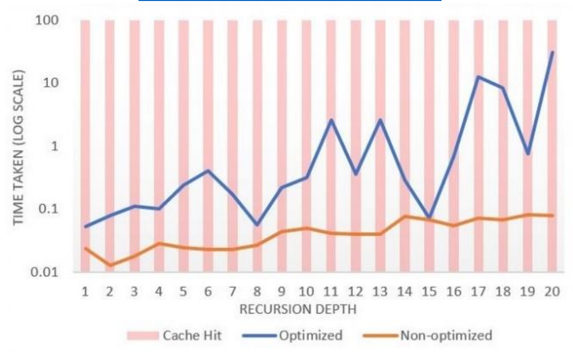


The function was tested till recursion depth of 20 once. The optimization being tested in this section performed orders of magnitude more poorly even though the cache hit rate was near perfect.



The function was tested till recursion depth of 20 once. The optimization being tested in this section performed significantly better, only worsening when there was a cache miss.

Caching symbolic variable for a



The function was tested till recursion depth of 20 once. The optimization being tested in this section performed poorly again when compared to the benchmark results.

Conclusion

The first optimization discussed, pertaining to the retention of learned clauses from previous stages, does offer promising improvements in efficiency. It should be noted, however, that use of various forms of caches should ensure this optimization is not weakened or ignored whenever queries to the solver are made.

The other two optimizations need to be further studied. Particularly, if its possible to enhance the existing solvers to accept partial assignments without losing the learned clauses.

Appendix

This section explains the C++ functions and their path constraints that were used to test the optimizations.

Simple Greatest Common Divisor (GCD) function

The function and the corresponding path constraints test are as follows:

```
// calculates the GCD
// a – signed int (4 bytes)
// b – signed int (4 bytes)
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

// constraint i represents recursion of depth at least i

1. $b \neq 0$
2. $b \neq 0 \,\wedge\, a\%b \neq 0$
3. $b \neq 0 \,\wedge\, a\%b \neq 0 \,\wedge\, b\%(a\%b) \neq 0$
4. $b \neq 0 \,\wedge\, a\%b \neq 0 \,\wedge\, b\%(a\%b) \neq 0 \,\wedge\, (a\%b)\%\big(b\%(a\%b)\big) \neq 0$
5. …

It is apparent how quickly the constraints are getting complex. This means the constraint solving time will be more reflective of the efficiency rather than process scheduling and other machine-dependent coincidences.

Controlled GCD Function

The function and the corresponding path constraints test are as follows:

```
// calculates the GCD till
recursion depth of at max c
// a – signed int (4 bytes)
// b – signed int (4 bytes)
// c – signed int (4 bytes)
int gcd(int a, int b, int c) {
    if (b == 0)
        return a;
    else if (c == 0)
        return b != 0 ?  0 : a;
    else
        return gcd(b, a % b, c - 1);
}
```

// constraint i represents recursion of depth at least i

1. $b \neq 0 \,\wedge\, c \neq 0$
2. $b \neq 0 \,\wedge\, c \neq 0 \,\wedge\, a\%b \neq 0 \,\wedge\, c-1 \neq 0$
3. $b \neq 0 \,\wedge\, c \neq 0 \,\wedge\, a\%b \neq 0 \,\wedge\, c-1 \neq 0 \,\wedge\, b\%(a\%b) \neq 0 \,\wedge\, c-2 \neq 0$
4. $b \neq 0 \,\wedge\, c \neq 0 \,\wedge\, a\%b \neq 0 \,\wedge\, c-1 \neq 0 \,\wedge\, b\%(a\%b) \neq 0 \,\wedge\, c-2 \neq 0 \,\wedge\, (a\%b)\%\big(b\%(a\%b)\big) \neq 0 \,\wedge\, c-3 \neq 0$
5. …

It adds another variable to our path constraints so that partial assignments can be tested on more combinations.

LOG Function

The function and the corresponding path constraints test are as follows:

```
// calculates the log base 2
// a – unsigned int (4 bytes)
unsigned int log(unsigned int a) {
    unsigned int pow = 0;
    while (a != 0) {
        a = a >> 1;
        ++pow;
    }
    return pow - 1;
}
```

// constraint i represents while-loop execution of at least i times

1. $a \neq 0$
2. $a \neq 0 \,\wedge\, (a \gg 1 \neq 0)$
3. $a \neq 0 \,\wedge\, (a \gg 1 \neq 0) \,\wedge\, (a \gg 2 \neq 0)$
4. $a \neq 0 \,\wedge\, (a \gg 1 \neq 0) \,\wedge\, (a \gg 2 \neq 0) \,\wedge\, (a \gg 3 \neq 0)$
5. …

This function generates much simpler constraints for SMT solvers, especially because only bitwise operations are used.

References

[1] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu and I. Finocchi, "A Survey of Symbolic Execution Techniques," *Computing Research Repository (CoRR),* vol. abs/1610.00502, 2016.

[2] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI),* pp. 209-224, 2008.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS),* pp. 322-335, 2006.

[4] W. Visser, J. Geldenhuys and M. B. Dwyer, "Green: Reducing, Reusing and Recycling Constraints in Program Analysis," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE),* no. 58, pp. 1-11, 2012.

[5] X. Jia, C. Ghezzi and S. Ying, "Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution," *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA),* pp. 177-187, 2015.

[6] D. A. Ramos and D. Engler, "Under-constrained Symbolic Execution: Correctness Checking for Real Code," *Proceedings of the 24th USENIX Conference on Security Symposium (SEC),* pp. 49-64, 2015.

[7] L. De Moura and N. Bjørner, "Z3: An Effcient SMT Solver," *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS),* pp. 337-340, 2008.

[8] L. Hadarean, *An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories,* New York University, 2015.