A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution

Tianhai Liu[1], Mateus Araújo[2], Marcelo d'Amorim[2], and Mana Taghdiri[1]

[1] Karlsruhe Institute of Technology, Germany
[2] Federal University of Pernambuco, Brazil

Abstract. Constraint solving is a major source of cost in Symbolic Execution (SE). This paper presents a study to assess the importance of some sensible options for solving constraints in SE. The main observation is that stack-based approaches to incremental solving is often much faster compared to cache-based approaches, which are more popular. Considering all 96 C programs from the KLEE benchmark that we analyzed, the median speedup obtained with a (non-optimized) stack-based approach was of 5x. Results suggest that tools should take advantage of incremental solving support from modern SMT solvers and researchers should look for ways to combine stack- and cache-based approaches to reduce execution cost even further. Instructions to reproduce results are available online: `http://asa.iti.kit.edu/130_392.php`

1 Introduction

Symbolic Execution (SE) [13,17,18,21,26] is a technique for systematic test-input generation that has gained significant momentum in recent years. Unfortunately, SE is expensive. It needs to explore many program paths and the execution of each path is more expensive compared to a non-symbolic (i.e., concrete) execution. Improving both aspects – space and time – is therefore important and a significant amount of research has been done in this direction recently [16]. The focus of this paper is on time reduction.

SE tools heavily use constraint solvers to avoid the exploration of infeasible paths and to generate test inputs; it comes with no surprise that constraint solving is often reported as the execution time sink of the technique [12,15,32,35].

Incremental solving is an important feature to address this high cost; it leverages the similarity across similar constraints to reduce overall solving cost. Intuitively, when using such feature, solving a set of similar constraints can be faster compared to solving each constraint in the set separately. Considering the fact that constraints that SE generates are similar by construction, existing SE tools employ some form of incremental solving to speedup execution.

1.1 Incremental Constraint Solving Approaches

One simple alternative to incremental solving is to only solve the "changed parts" of the constraint. For example, consider that SE produces the constraint $pc_1 : a>b \wedge x<y$ for which the solver outputs the following solution [a=2,b=1,x=3,y=4]. To compute the solution to the next constraint $pc_2 : a>b \wedge x\geq y$ this approach proceeds as follows: it invokes the solver to solve only the changed part of the constraint, namely $x\geq y$, which is a simpler problem, and combines the new solution [x=4,y=3] with the already-computed solution [a=2,b=1]. The combined solution clearly satisfies pc_2. This idea works under the assumption that SE explores similar paths in order (e.g., using depth-first search) and that not all variables in a constraint are dependent (e.g., x and y are mutually-dependent but do not depend on a and b).

The second alternative builds on the observation that the approach discussed above could be generalized to build on the solutions of all previously visited path constraints as opposed to only the last one visited. It caches solutions of every independent expression observed in every path constraint: two expressions are independent if they do not share any symbolic variables. Considering the previous example, a global cache stores solutions to the expressions $a>b$, $x<y$, and $x\geq y$ which appeared independently in the two individual path constraints pc_1 and pc_2. Despite the overhead in memory and time consumption related to caching (to store, lookup, and combine solutions), it has been observed that this optimization is beneficial. Popular symbolic execution tools, such as CREST [14], KLEE [15], PEX [31] and SPF [25], use similar features.

Another alternative to incremental solving makes use of built-in SMT solver support to solve similar constraints. It builds on the observation that as the paths that a SE explores gets longer chances of merging independent expressions increase since the number of input variables is limited. Unfortunately, the approaches to incremental solving presented above cannot help in this scenario. For example, the cached solution [x=3,y=2] to the constraint $x>y$ will not help to solve the constraint $x>y \wedge x>3$. In contrast, modern incremental SMT solvers, such as CVC4 [4], MathSAT5 [6], Yices [9], and Z3 [10] can help in this case: during constraint solving these tools learn lemmas, which can be later (re)used to solve similar *but not identical* constraints. To the best of our knowledge *no* existing SE tool uses such alternative for constraint solving.

1.2 Contribution

This paper reports the results of a study we conducted to assess how cache-based approaches compare with stack-based approaches to solve constraints incrementally. We considered various options of incremental solving and a large set of programs; both real (96 C programs from the KLEE [15] benchmark) and artificially-generated (300 randomly-generated programs of various sizes: 5, 10, and 20K). Overall, results indicate that stack-based approaches provide superior results. The median speedup obtained when using the support of a modern incremental SMT solver is of ∼5x (min.:∼1x, avg.:∼4.8x, max.: ∼9x).

In the light of these results, we investigated how to further improve stack-based approaches. We noticed that sharing of common expressions can facilitate the search for solution in SMT solvers [11]. We investigated the alternative of eliminating all common sub-expression from the constraint instead of of relying on the built-in heuristics from the solver. Results indicate that the speedup obtained with this alternative was of ∼1.11x over the benefits obtained with the basic stack-based incremental approach.

2 Background

Symbolic Execution is comprised of two parts: constraint generation and constraint solving. We briefly explain each part below. More details can be found elsewhere [16, 24, 27].

2.1 Constraint Generation

When symbolic execution evaluates a branch instruction, it needs to decide which branch of the control flow to select. In a regular concrete execution, the evaluation of a boolean expression is either true or false. Thus only one branch of the conditional can be taken. In contrast, in symbolic execution, the evaluation of a boolean expression is a symbolic value, so both branches can be taken resulting in different paths to be explored in the program. Symbolic execution characterizes each path it explores with a *path-condition* over the input variables $\overrightarrow{x}$. This condition is defined with a conjunction of boolean expressions $pc(\overrightarrow{x}) = \bigwedge b_i$. Each boolean expression b_i denotes a branching decision made during the execution of a distinct path in the program under test. Symbolic execution terminates when it explores all such paths corresponding to the different combinations of decisions. Programs with loops and recursion may result in an infinite number of paths; in those cases, one needs to define a bound on the number of paths that symbolic execution can explore.

2.2 Constraint Solving

Symbolic execution uses constraint solving (i) to check path feasibility and (ii) to generate test inputs. In the first case, symbolic execution checks if the current path is feasible by checking if the current (partial) path-condition is satisfiable. Exploration of a path is interrupted as soon as the path-condition becomes unsatisfiable. In the second case, symbolic execution uses a constraint solver to solve constraints associated with complete paths. The solutions to these constraints correspond to test inputs. SMT-LIB[3] is a popular format for describing constraints in SMT solvers [2, 4, 8, 10]. The SMT-LIB syntax [28] uses a prefix notation for expressions. For example, the user writes `(assert F)` to declare that a logical formulas F must hold. One can combine multiple formulas with logical operators. Symbolic names can be introduced as uninterpreted functions without arguments. Incremental SMT solvers [4, 6, 10] provide an assertion stack to solving similar constraints. The assertion stack is equipped with `push` and `pop` operations to enable one to keep contextual information. Each stack frame

[3]See `http://smtlib.cs.uiowa.edu`

```
int step(int a,int b){
  if (a < 0) b = a + b;
  if (b < 1) b = 2; else b = 3;
  return a+b;}
```

(a) Original

```
int stepOpt(int a,int b){
  if (a < 0) b1 = a+b; else b1 = b;
  if (b1 < 1) b2 = 2; else b2 = 3;
  return a+b2;}
```

(b) Transformed (for illustration)

Fig. 1. Sample code.

stores an *assertion set*, which includes locally-scoped declarations of functions, sorts, and logical formulas. The command `(check-sat)` holds if the conjunction of all assertion sets in the stack is satisfiable.

3 Techniques

We considered 5 techniques to evaluate effectiveness of cache-based and stack-based approaches to incremental solving. All techniques have been implemented in the same framework. We briefly describe them below.

- **Baseline** is the approach that does *not* use incremental solving. This approach conjoins all decisions reached along one path in a single constraint. That is, each constraint generated with SE results in a different potentially-long query to the solver.
- **Caching** refers to the technique that uses independent clauses optimization to simplify constraints before querying the solver (see Section 1.1). It incurs in overhead to partition constraints, lookup, and update the cache.
- **CachingOpt** optimizes *caching* by partitioning constraints incrementally. It keeps in memory the set of partitions and corresponding variables for the previously explored constraint. When reaching a control decision, it obtains new partitions by merging all partitions that have variables in common, considering the new variables involved in the decision. It incurs in additional overhead to merge partitions.
- **Stack** refers to the technique that creates a new frame on the assertion stack of an SMT solver when reaching a new control decision.
- **StackOpt** is as *stack* but builds constraints with new symbolic names so to facilitate identification of expression sharing.

3.1 Illustrative Example

This section illustrates the techniques. Figure 1 shows the code of function `step`; function `stepOpt` is obtained from code transformations on `step`. This function helps to illustrate the effect of common-subexpression elimination; new variables are introduced and each variable is defined only once. It is important to note that similar effect can be obtained without applying this transformation, e.g., by hash consing over assignment statements.

Figure 2 shows the SMT-LIB scripts produced by different techniques. In this example, each query terminates with the sequence of commands `check-sat` and `get-value` which indicate that the constraint was satisfiable and a solution could be retrieved. The solver context, that maintains the lemmas learned in previous computations, is destroyed with the command `exit`.

Figure 2(a) illustrates *Caching*. In this technique, each query starts with the construction of variables and terminates with the destruction of the solver context. For each query, only dependent constraints reach the solver; solutions are cached to avoid redundant queries. We show in comments the state of the cache and cache hit events. Figure 2(b) illustrates *Stack*. In this technique, the solver context evolves as new assertions are added to the stack; the context survives across the symbolic execution of different paths. To note that learned lemmas created on a stack frame are destroyed upon a `pop` of that frame.

3.2 Common Sub-expression Elimination: *StackOpt*

It is well known that sharing of structurally equal expressions can reduce space and time requirements in constraint solving, especially when dealing with large constraints. Modern SMT solvers identify those sharings automatically but there is cost associated with it and the mechanism to identify sharings is non-optimal.

Aware of that, we additionally evaluated the impact of translating the constraints to a representation that facilitates the identification of these sharings. In short, we eliminate common-subexpressions from input constraints. We want to evaluate how this feature works in conjunction with incremental SMT solving, which to the best of our knowledge is not used in these tools. We call the technique that uses this optimization *StackOpt*.

Consider, for example, the code fragment `if(.) a=x+y; if (a+z>10)` {.}. With traditional symbolic execution, the path corresponding to the traversal of the true branches is denoted by the constraint ... $x + y + z > 10$. *StackOpt*, however, translates this constraint to ... $a_1 = x_0 + y_0 \wedge a_1 + z_0 > 10$ as it identifies that the expression denoted by a_1 can be reused in other contexts. The use of such representation increases space requirements, i.e., it increases the number of variables and conjuncts in the constraint. On the other hand, it helps the constraint solver by letting it associate information with newly defined symbols (in this case, a_1).

Figure 2 shows side-by-side the scripts produced with this optimization disabled (*Stack*) and enabled (*StackOpt*). In contrast to *stack* and *caching*, that generate fresh constraints on decision points, *stackOpt* reuses expressions. For example, in Figure 2(c), *stackOpt* renames variable `b1` in query 1 to refer to `a+b`, and uses it in queries 2 and 3. We evaluate in this paper how such transformation can speedup stack-based constraint solving.

4 Evaluation

Our goal is to understand the extent to which constraint solving can be optimized for symbolic execution. We focused our attention to incremental solving, which

(a) Caching(Opt)

```
; query 1
(declare-fun a () Int)
(assert (< a 0))
(check-sat) ; sat
(get-value (a)) ; [a]:=[-1]
(exit)
; query 2
(declare-fun a () Int)
(declare-fun b () Int)
(assert (< a 0))
(assert (< (+ a b) 1))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [-1, 0]
(exit)
; query 3
(declare-fun a () Int)
(declare-fun b () Int)
(assert (< a 0))
(assert (not(<(+ a b) 1)))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [-1, 2]
(exit)
; query 4
(declare-fun a () Int)
(assert (not (< a 0)))
(check-sat) ; sat
(get-value (a)) ; [a]:=[0]
(exit)
; query 5
(declare-fun b () Int)
(assert (< b 1))
(check-sat) ; sat
(get-value (b)) ; [b]:=[0]
; cache hit : [!(a<0)]
; query 6
(declare-fun b () Int)
(assert (not (< b 1)))
(check-sat) ; sat
(get-value (b)) ; [b]:=[1]
; cache hit : [!(a<0)]
; cache: [a<0:[SAT,a:=-1],
; a+b<1:[SAT,a:=-1,b:=0],
; !(a+b<1):[SAT,a:=0,b:=2],
; !(a<0):[SAT,a:=0],
; b<1):[SAT,b:=0]]
```

(b) Stack

```
(declare-fun a () Int)
(declare-fun b () Int)
; query 1
(push)
(assert (< a 0))
(check-sat) ; sat
; query 2
(push)
(assert (< (+ a b) 1))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [-1, 0]
(pop)
; query 3
(push)
(assert(not(<(+ a b) 1)))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [-1, 2]
(pop)
(pop)
; query 4
(push)
(assert (not (< a 0)))
(check-sat) ; sat
;query 5
(push)
(assert (< b 1))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [0, 0]
(pop)
;query 6
(push)
(assert (not (< b 1)))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [0, 1]
(pop)
(pop)
(exit)
```

(c) StackOpt

```
(declare-fun a () Int)
(declare-fun b () Int)
; query 1
(push)
(assert (< a 0))
(check-sat) ; sat
(define-fun b1 () Int (+ a b))
; query 2
(push)
(assert (< b1 1))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [-1, 0]
(pop)
; query 3
(push)
(assert (not (< b1 1)))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [-1, 2]
(pop)
(pop)
; query 4
(push)
(assert (not (< a 0)))
(check-sat) ; sat
(define-fun b1 () Int b)
;query 5
(push)
(assert (< b1 1))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [0, 0]
(pop)
;query 6
(push)
(assert (not (< b1 1)))
(check-sat) ; sat
(get-value (a b))
; [a, b] := [0, 1]
(pop)
(pop)
(exit)
```

Fig. 2. SMT-LIB scripts produced with various techniques. Comments indicate what happens during exploration.

is the basic principle to solve large sets of similar constraints. We describe in the following the experiment we conducted to evaluate the techniques from Section 3.

4.1 Research Questions

We pose the following research questions.

RQ1. How cache-based and stack-based approaches compare?
RQ2. What is the benefit of using common sub-expressions elimination?

RQ3. Where each technique spends most time?

4.2 Objects of Analysis

We used two sets of programs in our evaluation. The first set includes programs automatically generated with RUGRAT [7]. The second set includes programs collected from the benchmark of KLEE [5], an open-source symbolic execution tool for C programs.

RUGRAT is a grammar-based program generator that has been proposed to support empirical evaluation of testing and analysis techniques. It produces programs based on weights associated to grammar production rules. A practical challenge for these kinds of generators is to construct realistic programs. However, an empirical study indicates that it is statistically impossible for a program analysis technique to differentiate a program written by a human from one that the tool generates [19]. The study compared real and generated programs with 78 existing software metrics. We considered three options of program size: programs of 5, 10, and 20 KLOC. We generated a total of 300 programs, 100 programs for each program size and only considered programs whose symbolic execution produces integer linear constraints.

The KLEE Coreutils benchmark [15] contains 96 Unix core programs (4.5 KLOC together). As the tool handles C programs we could not use our infrastructure (see Section 4.7). Instead, we ran KLEE, collected constraints produced by the tool, and analyzed them in order, i.e., consecutive constraints in the list reflect exploration order and are similar. For this reason we could not evaluate the combination *stackOpt* on this program set.

4.3 Experimental Variables

The *independent* variables of our experiment are the exploration time, size of the program, and exploration bounds. The *control* variables (i.e., constants) of our experiment are the choice of constraint solver and the exploration order. We used Microsoft's Z3 [10] for solving constraints and bounded depth first search for exploring paths. Even though results are deterministic we ran our scripts multiple times to confirm environmental changes did not introduce noise in our measurements. We used an Intel Xeon E5-2670 CPU with 2.60GHz clock running on a 64-bit openSUSE, and set 8GB as the max heap size for a symbolic execution.

4.4 RQ1. How cache-based and stack-based approaches compare?

To answer this research question we compared the effectiveness of techniques on the RUGRAT and KLEE benchmarks. We only considered variants without applying common sub-expression elimination in this experiment.

	5K	10K	20K
Solving Time (ms) per constraint			
baseline	8.100	16.750	25.565
caching	35.123	89.537	96.983
cachingOpt	17.547	45.624	47.630
stack	0.321	0.843	1.401
stackOpt	0.309	0.752	1.258

	5K	10K	20K
Number of queries answered			
baseline	29,154	9,115	5,856
caching	34,870	4,875	3,416
cachingOpt	58,988	10,097	5,047
stack	441,353	185,236	101,408
stackOpt	1177,907	448,545	256,345

Fig. 3. Cost metrics.

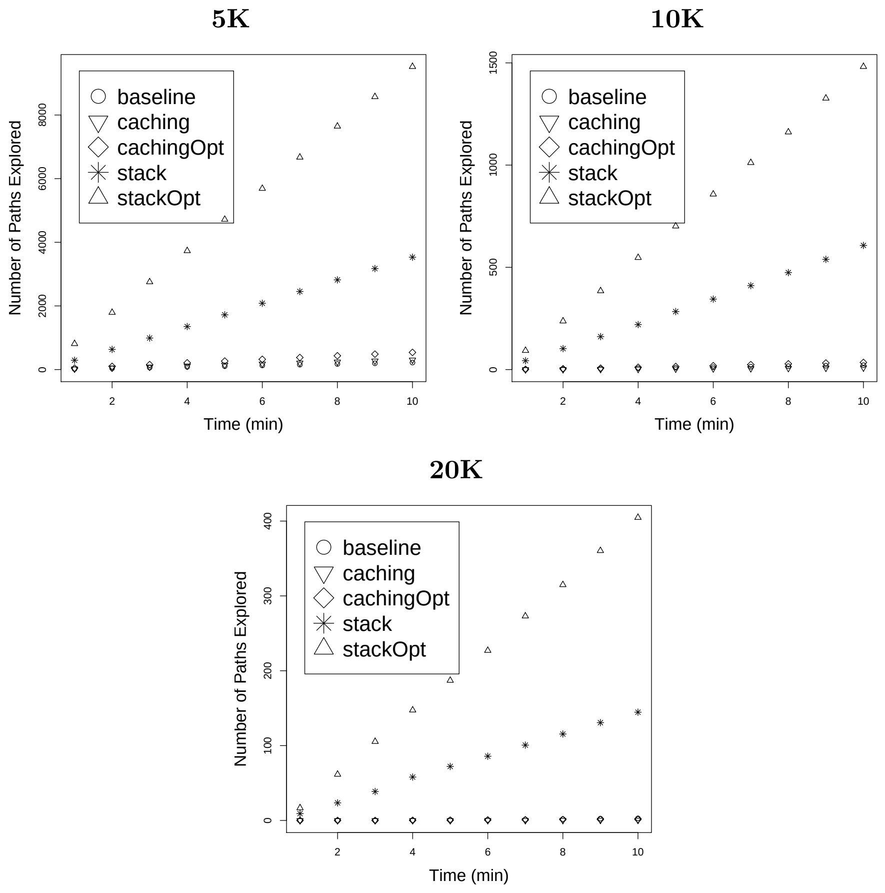


Fig. 4. Average number of complete paths explored (i.e., tests generated) using Z3. Time budget set to 10 minutes.

The RUGRAT **benchmark.** Figures 3 and 4 show results of various techniques for program generated with RUGRAT. We fixed the time budget for exploring paths in bounded depth-first order to 10 minutes.

Figure 3 shows the average cost for solving a constraint for each technique and the total number of queries answered. The average cost of solving one constraint is the total constraint solving time divided by the number of queries issued to the solver within a 10m time slot.

Each datapoint in the plots from Figure 4 indicates the number of explored complete paths (/tests generated) for a pair of technique and point in time. These plots show progress of different solving approaches. All plots indicate that the use of incremental SMT solving is beneficial. Note from the y-axis that as the size of programs grows the number of explored complete paths decreases. However, note that the speedups remain relatively similar. We observed that as size of programs grow constraint solving also become much more expensive; this justifies the decrease in number of complete paths explored on longer programs. Note from Figure 3 that the number of queries answered by the constraint solver in fact increase for longer programs compared to other techniques.

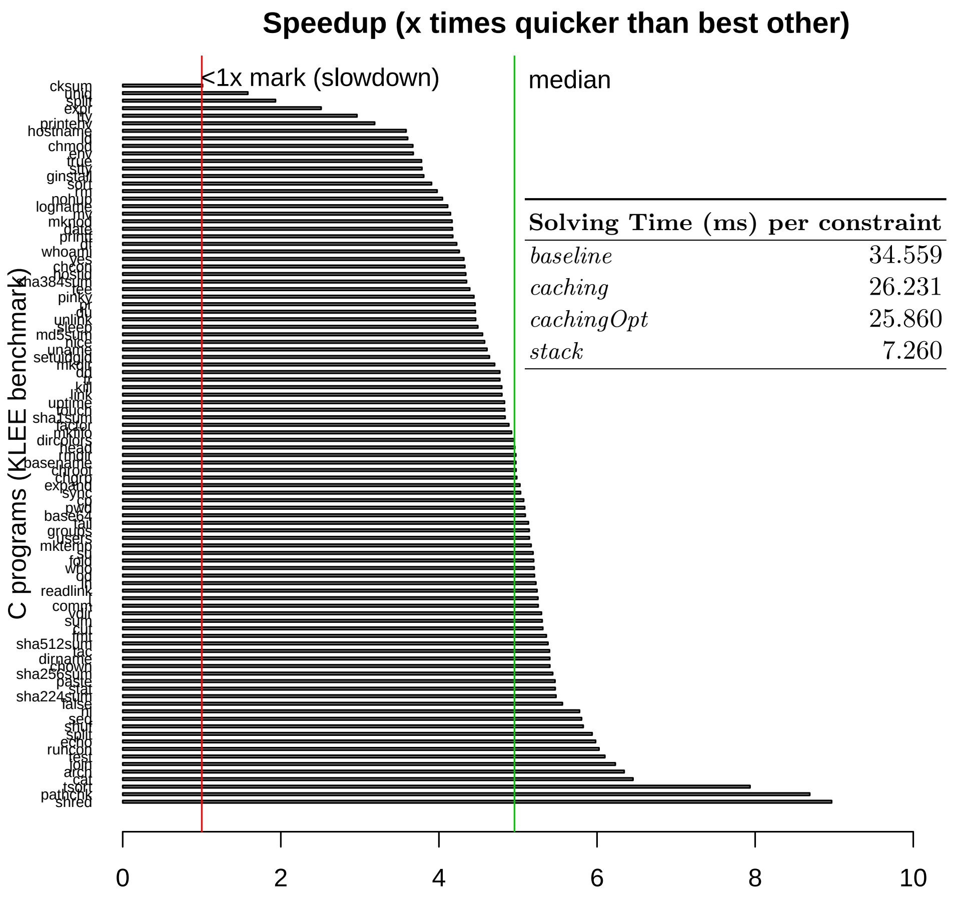
All plots from Figure 4 show a linear x-y relationship, indicating that the cost of exploring one path remains nearly the same during state-space exploration. Note that results are averaged across several programs. This linear behavior was a surprise. In principle, it would be justified only when feasible complete paths are uniformly distributed across the exploration tree and the cost of exploring one path is constant. A close inspection on results revealed that this indeed occurs many times although not always. However, as many subjects are considered a linear behavior emerged in the averaged plots.

The KLEE benchmark. We compared the techniques also using the benchmark of the SE tool KLEE [5, 15]. We analyzed the constraints it generates for 96 programs from KLEE's own benchmark. We set the time budget for SE to 30 seconds and used the default configuration for running KLEE. We confirmed as expected that KLEE spends most of its time budget (90%=∼27s/30s) in constraint solving[4]

Figure 5 shows the speedup that the best technique, *stack*, obtains compared to the second best technique, *cachingOpt*. The table in the right-top corner shows the time cost of solving each constraint. We did *not* evaluate *stackOpt* in this experiment as that would require post-processing KLEE-generated constraints. Considering the 96 programs analyzed the median speedup of *stackOpt* over *cachingOpt* was ∼5x. In absolute terms *stack* analyzed all constraint in 0.14s in the best case and 72.36s in the worst case, with a median cost of 6.3s and an average cost of 7.53s. For 91 of the 96 programs *stackOpt* was solved all constraints under 10s. 2 programs were solved under 30s and for only 2 programs it required more time: 54.9s and 72.36s.

It should be noted that the constraints from the KLEE benchmark build on the theory of bit-vectors whereas the constraints from the RUGRAT benchmark build on the theory of integers. We compared the techniques using different theories and obtained some evidence that the techniques we presented are effective for two relevant theories.

[4]http://klee.github.io/klee/klee-tools.html#klee-stats



Solving Time (ms) per constraint	
baseline	34.559
caching	26.231
cachingOpt	25.860
stack	7.260

Fig. 5. Speedup of stack-based incremental SMT solving over best alternative solving approach using Z3 (KLEE benchmark). The table in the right-top shows the solving time per constraint in various approaches.

4.5 RQ2. What is the benefit of using common sub-expression elimination?

Figure 4 shows that the *stackOpt* performs remarkably well. In contrast to *stack* this approach does not appear to degrade performance as the size of programs and constraints increase. The reason for the gain is justified: 1) On reaching each branch decision, *stackOpt* reuses the constraints constructed before path exploration while *stack* constructs new constraints when the variables involved in the branch condition were updated in the path leading to this branch. This is evidenced in Figure 4, in which *stack* has a notable overhead in path exploration. 2) To save search space and time, most modern SMT solvers (e.g., [1,2,4,6,9,10]) map structure-equal expressions to a singleton to construct a compact problem.

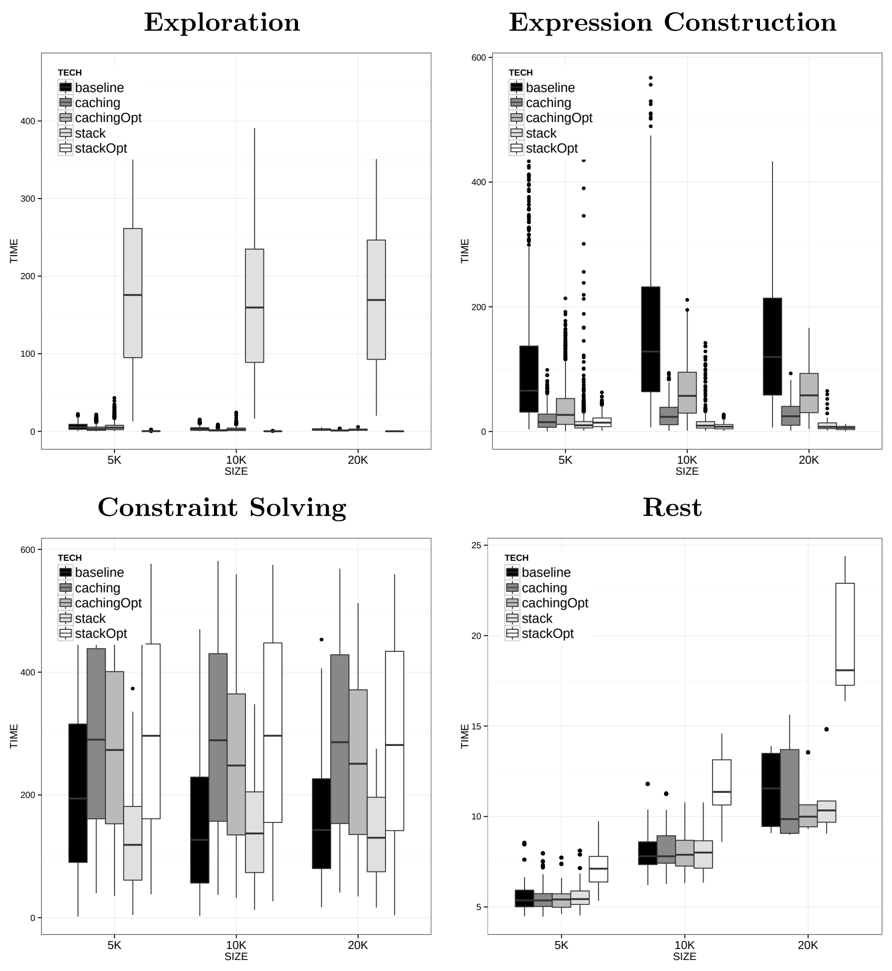


Fig. 6. Average time breakdown of different techniques using Z3 in 10 minutes.

While modern solvers detect shared expressions at the syntactical level, *stackOpt* introduces intermediate variables as macros to shared expressions at the semantic level.

4.6 RQ3. Where techniques spend most time?

Figure 6 shows the time breakdown of the techniques considering 4 sources of runtime cost: path exploration, expression construction, constraint solving, and rest. Path exploration time includes the cost of exploring paths (e.g., storing and restoring states), expression construction time includes the cost of creating Z3 expressions (we used Z3's programmatic interface for that), constraint solving time includes solving and and caching time, and rest includes the remaining parts, for example, the time of performing code transforms.

We make the following observations:

– *Baseline* spends more time in expression construction compared to other techniques. This happens because *baseline* needs reconstruct all Z3 expressions for a new query, while *caching* reduces the amount of constraints issued to the solver and consequently also reduces this cost.
– *Stack* spends more time in path exploration compared to other techniques. This happens because *stack* needs to update states on assignment statements and load states on decision points to generate fresh constraints, while *stack-Opt* has constraints constructed before path exploration. That is even worse for those paths traversed multiple times; *stack* will reload the states and recompute the constraints for each traversing, while *stackOpt* has constraints constructed prior to the path exploration.
– All caching techniques and *stackOpt* spent most time on solving constraints and at least 70% of the time is spent in constraint solving.
– *stack* spent less time in constraint solving compared to other techniques, while it can solve more constraints than any other technique except *stackOpt*.
– *stackOpt* spent more time in rest than other techniques. This happens is because *stackOpt* has a code transformation to rename variables.

4.7 Infrastructure

We developed a SE tool prototype to support our experiments. The motivation was to evaluate the influence of SSA. The infrastructure has been implemented in Java in ∼19.7KLOC, being∼1.5KLOC from InspectJ [23]. We computed non-blank non-comment lines of source code with the CLOC tool [3]. We used the Soot optimization framework [29] to process Java bytecodes, the Jung graph framework [20] to construct and explore decision graphs, and InspectJ to unroll loops and inline methods. The infrastructure generates constraints in SMTLIB v2 so it can interface with any compliant solver. For example, Z3 [10] is called directly through its programmatic interface to create corresponding Z3 expressions.The infrastructure supports both integers and bit-vectors to assess the impact of various options of incremental solving to speedup symbolic execution. The infrastructure reuses the created objects to reduce the cost of time and memory allocation in constraint generation.

Static Transformations. We implemented a sequence of static code transformations before the construction of decision graph. For example, unroll loops according to the configurations, model the program in Static Single Assignment (SSA) representation and inline methods on each call site. Finally, we obtained a directed acyclic graph with unique variable names. We evaluated how costly code transformation can be relative to the other costs. We observed that the linearization (i.e., inlining methods and unrolling loops) procedure is significantly

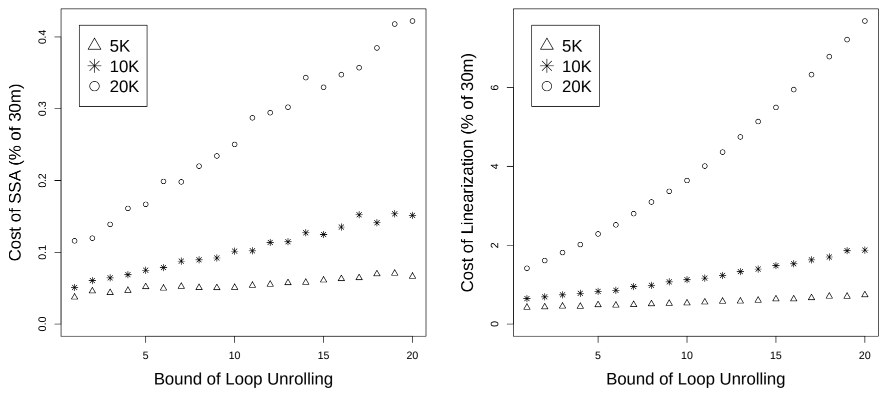


Fig. 7. Average percentage of static cost. For example, the average cost of linearization for a 20K program configured to unroll loops at most 3 times is approximately 34s(=(1.9/100)*30*60s)

more expensive compared to SSA, which is applied within each procedure prior to linearizing the code. But still linearization is relatively low cost. Figure 7 shows how each of these operations scale with program size and bound of loop unrollings. 60 subjects have been checked with a timeout 30m. The scale of the y-axis is the percentage of a 30m time budget. Results are averaged across all subjects considered for that size. In the worst case, linearization on 20K programs with 20 loop unrollings takes roughly 2m24s (=144s=8% of 30m).

4.8 Threats to Validity

As usual it is possible that results do not generalize much beyond our subject set. To mitigate this threat we used a set of 300 automatically-generated Java programs and a set of 96 real C programs from the GNU operating system.

Another threat to validity is the possibility of errors in our implementation. We carefully inspected our code and the consistency of our results. In summary, additional experiments are necessary to assess generality of our results.

5 Related Work

Symbolic Execution (SE) is expensive in time and space. We discuss below most-related recent work to reduce the high cost in constraint solving during SE.

5.1 Caching schemes

Cadar *et al.* [15] proposed several optimizations to simplify constraints prior to calling the solver during SE. The SE tool KLEE implements *caching* as we described. In addition, KLEE implements constraint checking with a potential solution. It is based on the assumption that a solution of subset often satisfy extra constraints. We remain to investigate how this additional optimization compares with those we considered.

Visser *et al.* [33] proposed GREEN, an infrastructure to share results of symbolic executions across different environments. GREEN proposes canonical representations of path conditions to enable caching across different programs. The intuition is that after partitioning constraints w.r.t. dependent clauses the chance of finding structurally equal symbolic constraints increases. For example, solutions to constraints produced in the symbolic execution of one program could be used to solve constraints produced from SE for another program. Results of GREEN are encouraging. Although the goal of GREEN is the same (to speedup constraint solving), our contributions are complementary.

5.2 Incremental SMT Solving

Incremental SMT solving is an active field of research with the goal of optimizing problems that can be characterized by many similar sub-problems. For example, detecting the program execution trace which maximizes execution cost [22], solving scheduling problems [30]. As a basic decision procedure, incremental SMT solving searches for a satisfying assignment by performing various operations (e.g. unit propagation). When internal conflicts occur incremental SMT solvers extract and store conflict clause to prune exploration search space. More specifically, incremental solvers store learned and conflicting clauses in the assertion stack so that they can be reused upon backtracking. Recently, Wieringa [34] proposed a technique to strengthen the clauses learned by the solver by extending an incremental SMT solver to execute in multiple threads. We observed that this development can directly improve symbolic execution.

6 Conclusions

This paper reports on a study to assess the impact of various options of incremental solving to speedup Symbolic Execution (SE). Results suggest that incremental solving is very important and that stack-based approaches provide superior results when compared to cache-based approaches for the benchmarks used in our experiments. Note that results are restricted to the use bounded depth-first search. More research is needed to find ways to combine caching- and stack-based approaches to improve results even further.

Acknowledgments. This work is partially funded by DFG grant TA764/1-1. Mateus is supported by the FACEPE fellowship # IBPG-0668-1.03/12. DFG is the German Research Foundation and FACEPE is the state of Pernambuco, Brazil, Research Foundation.

References

1. Alt-Ergo webpage. `http://alt-ergo.lri.fr/`.

2. Boolector webpage. `http://fmv.jku.at/boolector/`.
3. CLOC webpage. `http://cloc.sourceforge.net/`.
4. CVC4 webpage. `http://cvc4.cs.nyu.edu/web/`.
5. KLEE webpage. `http://klee.github.io/klee/`.
6. MathSAT5 webpage. `http://mathsat.fbk.eu/`.
7. RUGRAT webpage. `http://www.rugrat.ws/`.
8. STP webpage. `https://sites.google.com/site/stpfastprover/`.
9. Yices webpage. `http://yices.csl.sri.com/`.
10. Z3 webpage. `http://z3.codeplex.com/`.
11. Milan Bankovic. Argosmtexpression: an smt-lib 2.0 compliant expression library. In *workshop of the SAT*, June 2012.
12. Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *PLDI*, pages 123–132, 2014.
13. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution. In *International Conference on Reliable Software*, pages 234–245, 1975.
14. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. ASE '08, pages 443–446, 2008.
15. Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
16. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, pages 1066–1071, 2011.
17. Lori A. Clarke. A Program Testing System. In *ACM Annual Conference*, pages 488–491, 1976.
18. W. E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE TSE*, 3(4):266–278, 1977.
19. Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Chen Fu, Qing Xie, Sangmin Park, Kunal Taneja, and B. M. Mainul Hossain. Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator. In *WODA*, pages 1–6, 2012.
20. Jung webpage. `http://jung.sourceforge.net/`.
21. James C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.
22. Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with smt solvers. *SIGPLAN Not.*, 49(1):607–618, January 2014.
23. Tianhai Liu, Michael Nagel, and Mana Taghdiri. Bounded program verification using an smt solver: A case study. In *ICST*, pages 101–110, 2012.
24. Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, October 2009.
25. Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.
26. C. Ramamoorthy, S. Ho, and W. Chert. On the automated generation of program test data. *IEEE TSE*, 2(4):293–300, 1976.
27. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SP*, pages 317–331, 2010.

28. SMT-LIB webpage. `http://www.smtlib.org/`.
29. Soot webpage. `http://www.sable.mcgill.ca/soot/`.
30. Wilfried Steiner. An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks. In *RTSS*, pages 375–384, 2010.
31. Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *TAP*, pages 134–153, 2008.
32. Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *FSE*, pages 1–11, 2012.
33. Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.
34. Siert Wieringa. *Incremental Satisfiability Solving and its Applications*. PhD thesis, Aalto University, Finland, 2 2014.
35. Guowei Yang, S. Khurshid, and C. S. Pasareanu. Memoise: A tool for memoized symbolic execution. In *ICSE*, pages 1343–1346, May 2013.