Accelerating Invariant Generation

Kumar Madhukar*†, Björn Wachter‡, Daniel Kroening‡, Matt Lewis‡ and Mandayam Srivas†

*Tata Research Development and Design Center, Pune, Maharashtra, India
Email: kumar.madhukar@tcs.com
†Chennai Mathematical Institute, Chennai, Tamil Nadu, India
Email: mksrivas@hotmail.com
‡Department of Computer Science, University of Oxford, United Kingdom
Email: bjoern.wachter@cs.ox.ac.uk, kroening@cs.ox.ac.uk, matt@improbable.io

***Abstract*—Acceleration is a technique for summarising loops by computing a closed-form representation of the loop behaviour. The closed form can be turned into an *accelerator*, which is a code snippet that skips over intermediate states of the loop to the end of the loop in a single step.**

Program analysers rely on invariant generation techniques to reason about loops. The state-of-the-art invariant generation techniques, in practice, often struggle to find concise loop invariants, and, instead, degrade into unrolling loops, which is ineffective for non-trivial programs. In this paper, we evaluate experimentally whether loop accelerators enable *existing* program analysis algorithm to discover loop invariants more reliably and more efficiently. This paper is the first comprehensive study on the synergies between acceleration and invariant generation. We report our experience with a collection of safe and unsafe programs drawn from the Software Verification Competition and the literature.

I. Introduction

Consider the program in Fig. 1. It contains a simple assertion, which follows the while loop. An automated proof of safety for this assertion requires a technique that is able to discover the loop invariant $sn = sn + (n - i) * a$. State-of-the-art software model checkers either fail to prove the program or even if they do (for a bounded value of n), they do so by completely unwinding the loop, which does not scale for large n.

```
#define a 2

int main() {
  unsigned int i, j, n, sn = 0;
  j = i;
  while(i < n){
    sn = sn + a;
    i++;
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Fig. 1: Sample Safe Program

The simple recurrent nature of the assignments in the loop of program makes it amenable to *acceleration* [1]–[4].

This research was supported by ERC project 280053 (CPROVER).

Acceleration is a technique used to compute the effect of repeated iteration of statements. Specifically, the effect of k loop iterations in the example program is that the variable sn is increased by $k * a$. The idea is to replace, wherever possible, a loop with its closed form to obtain an equivalent accelerated program that is hopefully easier to verify.

Acceleration in the general case is, of course, as difficult as the original verification problem. Practical applications of acceleration are therefore typically restricted to particular special cases. For instance, Jeannet et al. [4] consider the case of deterministic linear loops over continuous variables. As there are very few cases in which the transitive closure is effectively computable, it is frequently not possible to obtain an accelerator that captures the behavior of the loop precisely. Thus, acceleration can be over-approximative (most references) or under-approximative (e.g. [5]). Acceleration frequently specialises in particular application domains, e.g., control software. Furthermore, acceleration techniques are frequently tuned to a particular analysis technique (e.g., abstract interpretation or predicate abstraction) that is applied subsequently.

The conjectures of this paper are: 1) accelerators support the invariant synthesis that is performed by program analysers, irrespective of the underlying analysis approach, and 2) analysers supported by acceleration not only do better than the original ones, they also outperform other state-of-the-art tools performing similar analysis. We aim to test these hypotheses by performing an evaluation over an extensive set of benchmarks and a variety of tools. Since all our benchmarks are C programs, we require an acceleration technique that is applicable to C programs and the fixed-width machine integers that they use. We use a template-based method published at CAV 2013 [5] to obtain the accelerators, and add them to the programs as additional paths. This transformation preserves safety i.e., the acceleration neither over- nor under-approximates. We are unable to pass the accelerated programs to all common off-the-shelf analysers, but we nevertheless compare with other tools in our experiments to quantify the advantage that acceleration provides over the state of the art.

Recall our example program. The program with accelerator added is given as Fig. 2. The instrumented code in Fig. 2 can be used instead of the original code for model checking state properties, as they have equivalent sets of reachable states.

```
int nondet_int();
unsigned nondet_unsigned();

#define a 2

int main(){
  unsigned int i, j, n, sn, k = 0;
  j = i;
  while(i < n){
    if(nondet_int()){ // accelerate
       k = nondet_unsigned(); sn = sn + k*a;
       i = i + k;
       assume(i <= n); } // no overflow
    else{ // original body
       sn = sn + a; i++; }
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Fig. 2: Program from Fig. 1 with accelerator

We observe that several model checkers that failing on the original program are able to verify the accelerated program successfully.

The core contribution of this paper is an *experimental study*, with the goal to validate our conjectures stated earlier. We quantify the benefit of accelerators when using commodity program analysers. We use two analysers in our experiments to substantiate the first claim (that accelerators aid existing analyzers). CBMC [6] is the model checker used in [7]; as a bounded analyser, it makes no attempt to infer invariants and is only able to conclude correctness if the program is shallow. IMPARA [8] is a C program verifier based on the LAWI-paradigm. IMPARA generates invariants using a very basic approach that relies on weakest preconditions, and does not employ a powerful interpolation engine.

Both IMPARA and CBMC are characterised by very weak invariant inference, and are thus expected to benefit substantially from acceleration. To relate the outcome to the best invariant generation techniques, towards validating our second claim, we include two other analysers: CPAchecker [9] and UFO [10]. These tools implement a broad range of invariant generation methods, including various abstract domains and interpolation. The comparison is performed on over 200 benchmarks, including those used in the Software Verification Competition 2015.

Although acceleration has successfully been combined with interpolation-based invariant construction [11], to the best of our knowledge, there has not been a thorough experimental study that quantifies the benefits of using it in tools that aim to prove correctness. While [5] did integrate acceleration within a framework where paths in the CFG were explored lazily with refinement, the emphasis of their experiments was to accelerate bug detection for unsafe programs. Recently, a loop over-approximation technique based on acceleration was proposed in [12] but this technique is not applicable to unsafe programs. Moreover, there is no refinement to eliminate spurious counterexamples arising from the over-approximation in [12]. The experiments in [7] focus on bounded model checking and do not include state-of-the-art interpolation-based tools.

The rest of the paper is organized as follows. The next section gives an overview of each of the tools used in our experiments and of the acceleration method from [5], [7] and its scope and restrictions. Section III contains experimental data and a discussion of the results.

II. OVERVIEW OF THE ANALYSIS TOOLS

We start this section with a brief informal introduction of the different tools used for our experiments.

UFO [10] combines the efficiency of abstract interpretation with numerical domains with the ability to generalize by means of interpolation in an abstraction refinement loop. UFO starts by computing an inductive invariant for the given program and checks if the invariant implies the given property. If the implication does not hold, UFO employs SMT solvers to check the feasibility of counterexample produced. If the error path is found to be infeasible, an interpolation technique guided by the results of an abstract interpretation is used to strengthen the invariant.

CPAchecker [9] is a tool and framework that aims at easy integration of new verification components. Every abstract domain, together with the corresponding operations, implements the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. The framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written in a concise and convenient way. CPAchecker uses MATHSAT as an SMT solver, and CSISAT and MATHSAT as interpolation procedures. It uses CBMC as a bit-precise checker for the feasibility of error paths, JAVABDD as the BDD package and provides an interface to an Octagon representation as well.

CBMC [6] is a bounded model checker for ANSI-C programs. It works by jointly unwinding the transition relation encoded in the given program and its specification, to obtain a first-order formula that is satisfiable if there exists an error trace. The formula is then checked using a SAT or SMT procedure. If the formula is satisfiable, a counterexample is extracted from the satisfying assignment provided by the SAT procedure. The tool also checks that sufficient unwinding is done to ensure that no longer counterexample can exist by means of *unwinding assertions*. This enables CBMC to prove correctness if the program is shallow.

IMPARA [8] extends the IMPACT algorithm to support asynchronous concurrent processes using an interleaved semantics. IMPARA, which analyses concurrent C programs with POSIX or Win32 threads, efficiently combines partial-order-reduction with the IMPACT algorithm. This paper highlights the benefits of combining IMPARA with acceleration for sequential programs.

The IMPARA algorithm returns either a safety invariant for a given program, finds a counterexample or diverges. To this

end, it constructs an abstraction of the program execution in the form of an *Abstract Reachability Tree* (ART), which corresponds to an unwinding of the control-flow graph of the program, annotated with invariants. To prove a program correct for unbounded executions, a criterion is needed to prune the ART without missing any error paths. A covering relation assumes this role.

The tool constructs an ART by alternating three different operations on nodes: EXPAND, REFINE, and CLOSE. EXPAND takes an uncovered leaf node and computes its successors along a randomly chosen thread. REFINE takes an error node v, detects whether the error path is feasible and, if not, restores a safe tree labeling. First, it determines whether the unique path π from the initial node to v is feasible by checking satisfiability of the transition constraints along π. If it is satisfiable, the solution gives a counterexample in the form of a concrete error trace, showing that the program is unsafe. Otherwise, an interpolant is obtained, which is used to refine the labels and update the cover relation. CLOSE takes a node v and checks if v can be added to the covering relation. As potential candidates for pairs to be a part of the covering relation, it only considers nodes created before v. This is to ensure a stable behavior, as covering in arbitrary order may uncover other nodes, which may not terminate.

A. Overview

The acceleration procedure used in this paper is based on the method described in [5]. This method relies on a constraint solver to compute the accelerators. We first provide an overview of the steps of the acceleration procedure, and subsequently provide additional detail. From a high-level perspective, the procedure implements the following steps:

1) Choose a path π through the loop body to be accelerated.
2) Construct a path $\overset{\rightsquigarrow}{\pi}$ whose behaviour under-approximates the effect of repeatedly executing π an arbitrary number of times.
3) The construction also generates conditions under which the acceleration is an under-approximation. These conditions are given in the form of two constraints – a *feasibility constraint*, which denotes the condition under which $\overset{\rightsquigarrow}{\pi}$ can be applied, and a *range constraint*, which constraints the number of iterations. These constraints are included as *assume* statements in $\overset{\rightsquigarrow}{\pi}$.
4) By construction, the assumptions and constraints in $\overset{\rightsquigarrow}{\pi}$ may contain universal quantifiers ranging over an auxiliary variable that encodes the number of loop iterations. The procedure uses a few simple techniques to eliminate these quantifiers that work under certain restrictions. The path is not accelerated if it is not able to eliminate the quantifiers.
5) Augment the control flow graph of the original loop body with an additional branch corresponding to $\overset{\rightsquigarrow}{\pi}$ with a non-deterministic choice in the branch.
6) The accelerated paths subsume some (or sometimes all) paths in the original program. The augmented loop structure generated in the previous step is analyzed to build a trace automaton that filters some of the redundant paths. The result of this step is used to generate a final program with fewer paths.

The acceleration procedure, after executing the above steps, produces an instrumented code with the modifications described in the last two steps. For a program with several loops, possibly nested, the acceleration procedure processes the loops one at a time, inside-out for nested loops. In our experiments we analyse the instrumented code that is produced, without further modifications. This process of acceleration may succeed, fail or time out. The last two outcomes imply that either a closed form solution with a given template does not exist or acceleration was unable to find one.

In the following, we give a few more details of the procedure, the form of the accelerated paths produced and explain the conditions under which the procedure works.

B. Accelerating Scalar Variables in a Path

For scalar variables, the acceleration is generated by fitting a particular polynomial template. If $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ is the vector of variables in π, then the accelerated assignment generated for each variable is represented by the following polynomial function:

$$\begin{aligned} f_{\mathbf{x}}(\mathbf{X}^{\langle 0\rangle}, n) \;\overset{\text{def}}{=}\; & \sum_{i=1}^{k} \alpha_i \cdot \mathbf{x}_i^{\langle 0\rangle} \\ + \;& \left(\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathbf{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)} \right) \cdot n \\ + \;& \alpha_{(2\cdot k+2)} \cdot n^2 \end{aligned}$$

Here, n is the number of loop iterations that are summarized, $\mathbf{x}_1^{\langle 0\rangle}, \ldots, \mathbf{x}_k^{\langle 0\rangle}$ are the initial values for the variables and the α_i with $0 \leq i \leq 2k+2$ are the unknown coefficients.

The acceleration for a path is performed in two steps. In the first step, the procedure solves for the coefficients α_i. This is done by considering only the assignments in the path π, i.e., by ignoring all the conditions, including the loop condition. This employs a combination of linear algebra techniques to first uniquely solve for the coefficients and then makes queries to SMT solver to inductively check that the generated polynomial for each variable is consistent with loop execution for an arbitrary number of iterations. If, for some $\mathbf{x}_i$, the inductive check fails, then it means there is no acceleration possible that fits the template.

In the second step the procedure considers the path with all the conditions, and generates the feasibility constraint, i.e., the condition under which the path is feasible. The feasibility constraint is essentially the negation of $wlp(\pi^n; false)$, where wlp is the weakest liberal precondition. Intuitively, a cumulative path π^n would be infeasible iff any intervening path π in the n-iteration cycle, starting from the state given by the accelerator, is infeasible. That is, π^n is infeasible if for any $j < n$ the first time frame of the suffix $\pi^{(n-j)}$ is infeasible (time frame refers to an instance of π in π^n). Thus, checking whether $wlp(\pi^n, false)$ holds is equivalent to checking if, for some j between 0

and n, $wlp(\pi, false)$ holds (after substituting every variable in π by its accelerated closed form expressions). Thus, the feasibility constraint for π^n will, in general, contain a universal quantifier ranging over the number of loop iterations. This can be eliminated if the predicate in the body of the formula is monotonic over the quantified parameter. The procedure reduces the monotonicity check in a conservative fashion to a SMT query by defining a *representing function* that returns the size of the set of states for which a predicate is false. No acceleration is performed if the monotonicity check fails.

C. Range Constraints

Since closed-form expressions and the derived feasibility constraints usually contain the number of iterations n in them, an overflow is likely to break the monotonicity requirement when bit-vectors or modular arithmetic are used. Also, since the behaviour of arithmetic over- or under-flow in C is not specified for signed arithmetic, we conservatively rule out all occurrences thereof in the accelerated path. This is done by adding range constraints in the form *assume* statements, which enforce that none of the arithmetic expressions that involve n overflow.

D. Accelerating Array Assignments

Acceleration of array assignments is challenging, as under-approximating closed-form solutions for them can often only be expressed by formulas that contain quantifier alternation (existential inside universal) ranging over the number of loop iterations and the domain (index) of the array. It has been shown in [5] that for array assignments of the form $a[x] := e$ such a quantifier pattern can be eliminated under the following sufficient conditions.

- There exist accelerated closed-form expressions for the index variable x and the expression e.
- The function f_x defining the closed-form solution for the index variable is linear in the number of loop iterations.

Under the above conditions one can derive a closed form representing an under-approximation of the array assignments.

E. Eliminating Redundant Paths using Trace Automata

The instrumentation of the accelerators described in the introduction preserves the unaccelerated paths in the program along with the newly added accelerated paths – for instance, the *else* branch in Fig. 2. Note that the added paths subsume some of the previously existing program paths.

The idea presented in [7] is to eliminate executions that are subsumed by some other execution of the program. For instance, taking the same accelerated path twice in a row is equivalent to taking it just once (for instance, in Fig. 2, executing the *if* block twice for values k_1 and k_2 is the same as executing it once with the value of k equal to $k_1 + k_2$ – which is possible because k is chosen non-deterministically in each iteration).

Similarly, taking the unaccelerated path immediately after taking the accelerated path is subsumed by taking the accelerated path just once (with the value of k being one more its previously chosen value, in Fig. 2). The elimination of these redundant paths is done by encoding the redundancies as a regular expression, which is then translated into a *trace automaton* [13]. When the accelerated program executes, the states in this automaton are also updated and it is ensured that this automata never reaches a *reject* state. An optimized version of the accelerated code for the running example is given in Fig. 3. This is achieved by introducing an auxiliary variable g that determines whether the accelerator was traversed in the previous iteration of the loop. This flag is reset in the non-accelerated branch, which, however, in our example is infeasible.

```
int nondet_int();
unsigned nondet_unsigned();

#define a 2
int main(){
  unsigned int i,j, n, sn, k = 0;
  bool g = *;
  j = i;
  while(i < n){
    if(nondet_int()){ // accelerate
       assume(!g);
       k = nondet_unsigned(); sn = sn + k*a;
       i = i + k;
       assume( i <= n); // no overflow
       g = true;}
    else{ // original body
       sn = sn + a; i++;
       g = false;}
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Fig. 3: Program from Fig. 2 with instrumented trace automaton

III. Experimental Results

A. Experimental Setup

We ran our experiments on a set of 201 benchmarks (138 safe, 63 unsafe) collected from the sources listed in [14] (published at CAV 2014) and SV-COMP 2015. We have eliminated examples that had syntax errors and the ones that were not supported by the accelerator (array examples, for instance). We compare the performance of UFO, CPAchecker, CBMC (with and without acceleration) and IMPARA (with and without acceleration). The unwinding depth used for experiments with CBMC was 100 for unaccelerated programs and 3 for accelerated programs. All experiments were run on a dual-core machine running at 2.73 GHz with 2 GB RAM, with a timeout limit of 60 seconds.

We elaborate on the benchmarks and the tools used to aid reproducibility. The benchmarks were collected from [15]–[17], the loops category in SV-COMP 2015 and the acceleration examples in the regression suite of CBMC (revision 4503). The tools used in the experiment were UFO (the SV-COMP 2014 binary), CPAchecker (release 1.3.4, with sv-comp14.properties as the configuration file), CBMC (built from revision 4503, used with Z3 as the decision procedure) and IMPARA (version

```
int main(void) {
  unsigned int x = 0;
  while (x < 268435454) {
    if(x < 65520){
      x++;
    } else {
      x += 2;
    }
  }
  assert(!(x % 2));
}
```

Fig. 4: A safe benchmark showing the need for acceleration.

0.2, used with MiniSat). The benchmarks, the exact commands used to invoke the tools, and the full results are available at http://www.cmi.ac.in/~madhukar/fmcad15.

Table I summarizes the performance of each of the tools. We record the number of safe instances reported as safe (*correct proofs*), the number of safe instances reported as unsafe (*wrong alarms*), the number of unsafe instances reported as unsafe (*correct alarms*), the number of unsafe instances reported as safe (*wrong proofs*), the number of instances which could not be decided by the tool (*no result*), the number of instances on which the tool reported the correct result in the least amount of time (*fastest*), the number of instances on which the tool was the only one to report the correct result (*unique*) and a score for each tool, calculated using the scoring scheme of SV-COMP 2015.[1]

B. Example

Before we discuss the results, we present an example to demonstrate the effectiveness of acceleration. Consider the safe example shown in Figure 4. All the tools involved in our experiments fail to prove this example safe. Even when the timeout is increased to 15 minutes, the tools still timeout. In general, one needs a loop invariant strong enough to prove the assertion outside the loop, to avoid unwinding the loop to the full. None of the tools were able to find such a loop invariant. Upon acceleration, a closed form for the variable x is generated: $x = 1 * k + 2 * l$. The additional constraint generated for k, that $k = 65520$, along with the closed form for x (and negation of the loop termination condition) is strong enough an invariant to prove the property.

In some circumstances, acceleration uses quantifiers in the accelerated programs. These are not the ones arising from the feasibility or range constraints that we discussed in Section II (those get eliminated during the acceleration). These quantifiers appear while encoding the overflow constraints in the accelerated program. Suppose we want to construct a closed form for a variable being modified in a loop, by assuming that the loop executed i times. In this case, we need to assure that there is no overflow that was caused during any of these i iterations. In some cases, it is sufficient to assume that i^{th} iteration does not lead to an overflow. An instance is example 4, as the loop condition is $(x < 268435454)$. Thus, if the i^{th} iteration does not lead to an overflow, none of the previous iterations do. However, if we change the loop condition to $(x \neq 268435454)$ this does not hold any more. Therefore, it must be ensured separately for every $k \in [0, \ldots, i]$ that there is no overflow after k iterations. In our experiments, there were 40 benchmarks (roughly 25 %) that use quantifiers in their corresponding accelerated programs. The presence of quantifiers makes the verification task difficult as none of the tools is able to instantiate the quantifiers correctly. More effective quantifier handling will yield further results in favor of acceleration.

[1] Score = (2·*correct proofs*)−(12·*wrong proofs*)+*correct alarms*−(6·*wrong alarms*)

C. Discussion of Results

IMPARA + Acceleration clearly outperforms IMPARA without acceleration, UFO and CPAchecker. This underlines the benefit of acceleration as an auxiliary method for invariant generation. Note that we see an increase in the number of *correct proofs* as well as *correct alarms*. CPAchecker comes close in terms of the *correct proofs*, which we credit to its broad portfolio of techniques for generating invariants, including interpolation, abstract interpretation and predicate abstraction. The *wrong proofs* CPAchecker generates are partly caused by missing overflow situations.

When compared to CBMC + Acceleration, IMPARA + Acceleration does better for the following reason: The accelerators themselves are not helpful to CBMC for generating proofs – it simply unwinds the program CFG and makes a single decisive query to the solver. A large number of our benchmarks are safe, and CBMC only benefits from accelerators if the trace automaton is able to prune the original paths. By contrast, even without trace automata, acceleration may improve convergence of IMPARA, as acceleration can lead to "better" interpolants. Without acceleration an interpolation procedure is presented an unwinding of the loop body. It is well-known, see e.g. [18], that this can lead to overly specific interpolants that rule out only this particular unwinding. By contrast, in the accelerated program, the interpolation procedure is presented with the transitive closure of the loop; it thus is forced to compute an interpolant for a much larger number of unwindings. For instance, IMPARA without acceleration fails to generate a loop invariants for Figure 5, and thus falls back to loop unwinding, whereas, on the accelerated program, unwinding is avoided, and the tool generates the invariant $x + y = n$.

The overall score drops when combining CBMC with acceleration. This is due to the wrong alarms generated by the combination, which is heavily penalized according to the scoring rules at SV-COMP. There is a substantial increase in the number of correct proofs and correct alarms, however. The advantages of combining acceleration with CBMC and IMPARA (note that CBMC and IMPARA are very different tools) strongly suggests that a similar advantage could be obtained with other tools as well. An investigation of the cause for the increase in

TABLE I: Comparison of tools

Tools	Number of instances							Score
	correct proofs	wrong proofs	correct alarms	wrong alarms	no results	fastest	unique	
CPAchecker 1.3.4	83	16	35	14	53	18	11	−75
UFO SV-COMP 2014	52	2	18	2	127	4	2	86
CBMC r4503	32	0	35	0	134	16	1	99
+ Acceleration	53	0	45	12	91	28	9	79
IMPARA 0.2	78	1	36	15	71	73	0	90
+ Acceleration	86	0	47	12	56	36	6	147

```
int main(){
  unsigned int n = nondet_uint();
  int x = n;
  int y = 0;

  // loop invariant: x + y == n
  while(x > 0){
    x = x - 1;
    y = y + 1;
  }
  assert(y == n);
}
```

Fig. 5: Acceleration can improve generalisation in LAWI.

number of wrong alarms for CBMC and a precise quantification of the benefit of combining other tools is future work.

The fact that acceleration helps CBMC and IMPARA on unsafe instances is unsurprising; the technique we use was designed to aid counterexample detection [5]. The experimental results confirm that in addition, acceleration helps to generate invariants. Invariant generation techniques, in practice, often struggle to find concise loop invariants, and, instead, degrade into unrolling loops completely, which leads to poor performance and defeats the purpose of invariant generation. Our experiments demonstrate that there is a synergy between the two techniques, i.e., acceleration leads to better invariants, and invariant generation also helps finding bugs faster. We conjecture that the invariants steer the search for the bug away from irrelevant parts of the state space.

While CPAchecker employs a bit-accurate tool – by default CBMC – to verify counterexamples, its invariant-generation engine works over mathematical integers, i.e., invariants may hold over mathematical integers but are not checked with respect to integer overflow. Wrong proofs observed with CPAchecker mainly arise from deriving mathematical-integer invariants that do not hold in presence of overflow. In such situations, acceleration cannot help. This can be explained as follows. The accelerator represents a transitive closure of the loop body. It follows easily by induction that the result of CPAchecker, if the tool terminates, must be the same as for the unaccelerated program, since both programs are semantically equivalent.

The Appendix gives the complete results of our experiments. If a tool worked on a given benchmark and produced the expected result, we report the time taken by the tool in seconds. The entries to, inc, err and nr indicate, respectively, that the tool timed out, produced an incorrect result, terminated with an error or could not decide whether the input benchmark is safe or unsafe. The winning entry (in terms of the time taken by the tool) for each row (if there is one) is given in bold font. Note that the time taken by CBMC + Accleration and IMPARA + Accleration does not include the time taken to generate the instrumented program with accelerators. The latter is given separately in the column *Accl*.

IV. CONCLUSION AND FUTURE WORK

In this paper we have quantified the benefit of acceleration for checking safety properties. We report the results of a comprehensive comparison over a number of benchmarks, which shows that the combination of acceleration and a safety checker indeed outperforms existing techniques. The performance enhancement is visible for both safe and unsafe benchmarks, shown by an increase in the number of correct alarms as well as the correct proofs reported by the tool.

The source-level transformation of programs enables integration with futher invariant generation techniques. As a future work, we plan to investigate the interplay between acceleration and invariant generation to minimize the number of wrong alarms and to handle more cases correctly, including those that involve arrays. We also believe it would be worthwhile to investigate whether the accelerator can be assisted with additional invariants generated using some other technique. Our initial experiments suggest that some of these invariants, even over the interval domain, may help us rule out the possibility of overflows, thereby increasing the precision of the accelerator.

REFERENCES

[1] B. Boigelot, *Symbolic Methods for Exploring Infinite State Spaces*, ser. Collection des publications. Université de Liège, Faculté des sciences appliquées, 1999.

[2] M. Bozga, R. Iosif, and F. Konecný, "Fast acceleration of ultimately periodic relations," in *Computer Aided Verification*, ser. LNCS. Springer, 2010, vol. 6174, pp. 227–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_23

[3] A. Finkel and J. Leroux, "How to compose Presburger-accelerations: Applications to broadcast protocols," in *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, ser. LNCS. Springer, 2002, vol. 2556, pp. 145–156. [Online]. Available: http://dx.doi.org/10.1007/3-540-36206-1_14

[4] B. Jeannet, P. Schrammel, and S. Sankaranarayanan, "Abstract acceleration of general linear loops," in *Principles of Programming Languages (POPL)*. ACM, 2014, pp. 529–540. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535843

[5] D. Kroening, M. Lewis, and G. Weissenbacher, "Under-approximating loops in C programs for fast counterexample detection," in *Computer Aided Verification (CAV)*. Springer, 2013, pp. 381–396. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_26

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2004, vol. 2988, pp. 168–176. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24730-2_15

[7] D. Kroening, M. Lewis, and G. Weissenbacher, "Proving safety with trace automata and bounded model checking," in *Formal Methods (FM)*, ser. LNCS, vol. 9109. Springer, 2015.

[8] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with Impact," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 210–217. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679412

[9] D. Beyer and M. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification (CAV)*, ser. LNCS. Springer, 2011, vol. 6806, pp. 184–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_16

[10] A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik, "UFO: Verification with interpolants and abstract interpretation," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS. Springer, 2013, vol. 7795, pp. 637–640. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_52

[11] H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer, "Accelerating interpolants," in *Automated Technology for Verification and Analysis (ATVA)*, ser. LNCS. Springer, 2012, pp. 187–202. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33386-6_16

[12] P. Darke, B. Chimdyalwar, R. Venkatesh, U. Shrotri, and R. Metta, "Over-approximating loops to prove properties using bounded model checking," in *Design, Automation & Test in Europe (DATE)*. EDA Consortium, 2015, pp. 1407–1412. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757139

[13] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *Static Analysis (SAS)*, ser. LNCS. Springer, 2009, vol. 5673, pp. 69–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03237-0_7

[14] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, "Counterexample to induction-guided abstraction-refinement (CTIGAR)," in *Computer Aided Verification (CAV)*, ser. LNCS. Springer International Publishing, 2014, vol. 8559, pp. 831–848. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_55

[15] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, "Dagger Benchmarks Suite," http://www.cfdvs.iitb.ac.in/~bhargav/dagger.php, 2014.

[16] A. Gupta and A. Rybalchenko, "InvGen Benchmarks Suite," http://pub.ist.ac.at/~agupta/invgen/, 2014.

[17] A. Albarghouthi and K. McMillan, "Beautiful interpolants," in *Computer Aided Verification*, ser. LNCS, 2013, vol. 8044, pp. 313–329. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_22

[18] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Path invariants," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 300–309. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250769

APPENDIX
EXPERIMENTAL RESULTS

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
28.c	nr	to	nr	nr	to	to	1.27
25.c	nr	inc	to	**0.69**	to	to	0.52
20.c	nr	1.41	7.01	6.54	**0.02**	0.03	1.17
09.c	nr	to	nr	nr	to	to	1.81
05.c	nr	to	nr	nr	to	to	2.12
f2.c	to	inc	to	40.46	to	**10.17**	1.65
xyz2.c	nr	inc	nr	nr	to	**0.79**	1.14
xy0.c	nr	inc	nr	nr	to	**0.19**	0.53
gulv.c	to	47.18	nr	to	**38.83**	to	3.6
substring1.c	0.35	to	nr	**0.08**	42.79	0.26	0.26
24.c	nr	48.31	to	nr	**3.52**	to	0.49
xy4.c	nr	inc	nr	nr	to	**0.21**	0.68
pldi082_unbounded.c	nr	to	nr	nr	to	to	0.78
15.c	nr	14.65	nr	**0.32**	to	0.64	0.48
gulv_simp.c	**0.57**	46.13	nr	nr	to	to	1.14
33.c	nr	**47.62**	to	to	to	to	2.59
xy10.c	nr	19.78	0.21	0.35	**0.01**	0.02	0.45
xyz.c	nr	inc	nr	nr	to	**17.08**	1.18
12.c	nr	to	nr	nr	to	to	5.59
31.c	nr	inc	to	**1.75**	to	to	0.18
35.c	nr	inc	nr	**0.22**	to	0.35	0.29
07.c	nr	inc	nr	**0.06**	to	0.1	0.26
39.c	0.2	1.22	**0.01**	err	0.02	err	0.5
19.c	nr	inc	nr	**0.62**	to	to	0.83
37.c	nr	inc	nr	**0.24**	to	0.7	0.62
simple_safe1.c	0.21	to	nr	0.06	**0.01**	0.02	0.23
diamond_unsafe2.c	nr	1.45	9.44	**0.4**	0.53	0.84	0.6
underapprox_unsafe1.c	0.2	1.23	0.02	nr	**0.02**	0.03	0.38
nested_safe1.c	**0.2**	to	nr	nr	to	to	0.87
diamond_safe2.c	nr	17.34	2.29	**0.2**	0.43	0.85	0.57
const_unsafe1.c	0.26	1.5	**0.02**	0.06	0.03	0.03	0.2
functions_safe1.c	0.2	to	nr	0.06	**0.02**	0.03	0.23
nested_unsafe1.c	0.27	to	nr	nr	0.52	**0.09**	0.2
multivar_unsafe1.c	0.32	1.39	0.31	0.1	**0.01**	0.02	0.27
underapprox_safe2.c	0.2	1.39	0.02	nr	**0.02**	0.05	0.37
simple_unsafe1.c	0.2	to	nr	0.07	to	**0.03**	0.23
underapprox_unsafe2.c	0.2	1.49	0.03	nr	**0.02**	0.05	0.39
simple_safe4.c	0.2	to	nr	nr	0.02	**0.02**	0.2
simple_unsafe4.c	to	to	nr	nr	to	to	0.21
simple_unsafe2.c	0.39	1.73	0.17	0.07	**0.01**	0.01	0.22
phases_unsafe1.c	to	to	nr	**0.1**	to	0.63	0.33
diamond_safe1.c	nr	to	0.38	**0.11**	3.99	1.09	0.3
diamond_unsafe1.c	nr	2.49	0.31	0.16	54.33	**0.04**	0.34
simple_safe2.c	0.21	1.53	nr	0.07	**0.01**	0.02	0.19
const_safe1.c	0.21	1.28	**0.01**	0.04	0.02	0.03	0.21
overflow_unsafe1.c	0.2	inc	nr	**0.06**	to	0.11	0.19
simple_unsafe3.c	nr	1.42	0.08	0.08	**0.01**	0.01	0.23
phases_safe1.c	to	to	nr	**0.12**	to	0.93	0.29
simple_safe3.c	0.16	to	nr	0.06	**0.01**	0.03	0.24
multivar_safe1.c	nr	1.54	nr	0.07	**0.01**	0.04	0.28
functions_unsafe1.c	0.2	to	nr	0.06	to	**0.02**	0.22
underapprox_safe1.c	0.19	1.4	0.02	nr	**0.02**	0.04	0.38
overflow_safe1.c	0.21	47.8	nr	0.08	**0.01**	0.03	0.22
efm.c	nr	inc	to	to	inc	err	3.76
hsortprime.c	nr	**51.33**	to	nr	inc	to	1.03
bk-nat.c	nr	inc	25.61	39.74	**0.43**	err	3.38
barbrprime.c	nr	1.78	to	12.41	**0.06**	0.65	5.21
fig1a.c	nr	to	nr	nr	to	to	1.03
swim.c	nr	**48.9**	to	to	to	err	4.2
seesaw.c	nr	**47.73**	to	nr	to	err	1.59
swim1.c	nr	**52.44**	to	to	inc	inc	4.7
barbr.c	nr	**1.85**	to	22.63	8.38	55.01	7.26
ex1.c	nr	inc	**42.07**	nr	to	to	1.62
cars.c	nr	to	to	to	to	to	13.33
fig2.c	inc	inc	to	24.89	to	**11.54**	1.59
lifo.c	nr	**19.61**	to	to	29.98	40.92	9.43
lifnatprime.c	nr	inc	to	to	inc	**34.76**	8.31
ex2.c	nr	7.07	**0.04**	0.05	8.03	8.41	0.09
bkley.c	nr	inc	23.16	**15.03**	to	err	2.99
hsort.c	nr	inc	to	nr	inc	inc	1.43
lifnat.c	nr	inc	to	to	inc	**58.74**	9.73
seq-len.c	nr	to	to	nr	to	to	1.57

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
svd-some-loop.c	nr	**2.42**	to	nr	inc	5.74	2.2
split.c	nr	to	nr	nr	to	to	0.12
string_concat-noarr.c	0.21	1.25	0.01	0.18	**0.01**	0.03	0.64
bind_expands_vars2.c	nr	19.08	nr	**1.39**	to	2.42	0.37
simple_if.c	nr	**37.98**	nr	nr	to	to	0.45
nest-if5.c	0.2	19.93	nr	nr	0.01	**0.01**	0.38
up-nested.c	0.21	1.36	nr	nr	**0.01**	0.02	0.25
NetBSD_g_Ctoc.c	0.2	1.22	0.01	0.02	**0.01**	0.02	0.93
nested8.c	nr	**22.81**	to	nr	inc	60.0	1.95
nest-len.c	nr	48.98	to	nr	to	**7.33**	0.76
nested2.c	0.59	47.6	to	nr	**0.01**	0.03	0.27
heapsort3.c	nr	inc	to	nr	inc	inc	0.33
sendmail-close-angle.c	nr	to	nr	**1.2**	to	inc	0.91
NetBSD_glob3_iny.c	0.26	1.26	0.01	err	**0.01**	err	0.55
nested.c	0.2	47.03	to	nr	to	**0.1**	0.24
seq-sim.c	nr	to	to	nr	to	to	1.06
puzzle1.c	nr	1.36	to	nr	0.01	**0.01**	0.63
half.c	nr	to	nr	**0.76**	to	4.28	1.53
MADWiFi-encode_ie_ok.c	nr	18.98	15.1	0.28	**0.06**	0.14	0.41
simple.c	nr	inc	nr	0.12	**0.02**	0.14	0.15
nested1.c	0.63	47.95	to	nr	**0.01**	0.02	0.25
mergesort.c	nr	1.57	0.02	0.01	0.02	**0.01**	1.1
svd4.c	nr	**3.87**	to	err	inc	inc	6.09
spin.c	0.2	1.26	0.01	0.01	0.01	**0.01**	0.06
svd2.c	nr	1.51	to	nr	**0.01**	0.03	0.74
spin1.c	0.21	1.24	0.01	0.01	0.01	**0.01**	0.07
heapsort.c	nr	inc	to	nr	inc	inc	1.32
nest-if7.c	0.2	48.21	to	nr	**0.09**	0.13	0.48
sendmail-mime7to8_arr_three_chars_no_test_ok.c	0.2	2.75	to	1.58	0.02	**0.01**	0.31
nest-if1.c	nr	inc	to	nr	**0.29**	0.84	0.25
simple_nest.c	nr	1.35	nr	nr	**0.06**	0.29	0.81
NetBSD_loop_int.c	0.2	1.25	0.01	err	**0.01**	err	0.34
nested6.c	nr	47.26	to	nr	**7.88**	8.07	0.5
down.c	nr	to	nr	**0.37**	to	1.3	0.58
seq.c	nr	1.41	7.26	1.98	**0.02**	0.04	1.59
seq3.c	nr	to	to	nr	to	to	1.18
nested3.c	nr	inc	to	nr	to	to	0.39
nest-if2.c	nr	inc	to	nr	20.95	**0.8**	0.39
seq-proc.c	inc	to	to	**2.98**	to	to	1.24
nest-if4.c	0.2	19.91	to	nr	**0.01**	0.03	0.27
apache-escape-absolute.c	nr	to	to	nr	**0.21**	3.34	3.68
bound.c	nr	inc	nr	1.87	**0.02**	0.12	1.76
nest-if.c	0.6	47.82	to	nr	to	**0.13**	0.26
svd3.c	nr	1.58	to	nr	**0.02**	0.03	0.38
up5.c	nr	to	nr	nr	to	to	0.87
heapsort2.c	nr	**1.81**	to	nr	to	to	0.23
NetBSD_loop.c	nr	18.68	4.96	0.21	**0.01**	0.02	0.42
nested9.c	nr	to	to	nr	inc	inc	0.39
nested4.c	nr	nr	to	nr	to	to	0.38
up3.c	nr	to	nr	nr	to	to	1.03
heapsort1.c	nr	**1.79**	to	nr	to	to	0.25
SpamAssassin-loop.c	nr	18.84	to	err	**0.01**	0.05	1.65
seq2.c	inc	to	to	nr	to	inc	0.98
up.c	nr	to	nr	**0.34**	to	1.25	0.54
nest-if3.c	nr	inc	to	nr	**0.21**	0.64	0.17
apache-get-tag.c	nr	to	to	**7.88**	to	to	1.48
seq-z3.c	nr	to	to	**2.9**	to	to	1.22
fragtest_simple.c	0.2	1.22	**0.01**	0.52	0.02	0.12	0.83
rajamani_1.c	0.2	1.26	0.01	0.16	**0.01**	0.03	5.13
nest-if8.c	nr	47.64	to	nr	inc	**0.02**	0.35
sendmail-mime-fromqp.c	0.21	1.71	0.04	0.05	0.02	**0.02**	0.38
gulwani_cegar2.c	0.2	14.62	nr	0.2	**0.03**	0.95	0.26
test.c	0.2	1.3	0.02	0.01	0.01	**0.01**	0.01
nested7.c	nr	inc	to	nr	inc	inc	0.98
id_build.c	nr	to	to	nr	**0.11**	0.3	0.42
svd1.c	nr	**3.22**	to	nr	inc	5.65	4.35
id_trans.c	0.43	18.43	29.26	0.69	**0.01**	0.02	0.55
nested5.c	nr	48.22	to	nr	**0.08**	to	0.23
gulwani_cegar1.c	nr	inc	0.03	0.21	**0.02**	0.04	0.43
ken-imp.c	**0.19**	1.66	nr	0.55	to	0.68	0.42
sort_instrumented.c	0.33	1.26	0.01	0.01	0.01	**0.01**	0.73
SpamAssassin-loop_ok.c	0.61	47.11	to	nr	**0.02**	0.08	0.7
up-nd.c	inc	to	26.74	1.87	**0.07**	1.17	0.95
compact_false.c	to	to	nr	to	to	**21.47**	0.29
veris.c_OpenSER__cases1_stripFullBoth_arr_true.c	0.2	19.69	to	nr	**0.02**	0.05	0.99

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
terminator_02_true.c	nr	1.59	1.34	0.16	**0.01**	0.03	0.38
bubble_sort_false.c	nr	**6.89**	to	err	err	err	1.96
n.c11_true.c	nr	1.28	nr	nr	**0.02**	inc	0.2
ludcmp_false.c	0.22	to	err	err	**0.1**	err	0.12
trex02_false.c	nr	1.36	0.39	0.16	0.02	**0.01**	0.37
matrix_true.c	nr	7.48	**0.04**	nr	err	err	0.62
vogal_false.c	nr	**1.61**	11.03	to	err	31.52	1.44
trex03_false.c	nr	1.7	8.13	0.06	**0.01**	0.02	0.85
while_infinite_loop_2_true.c	0.2	1.29	nr	0.03	**0.01**	0.02	0.2
insertion_sort_true.c	nr	to	to	nr	to	inc	0.51
matrix_false.c	nr	**22.86**	to	to	err	err	0.74
verisec_OpenSER__cases1_stripFullBoth_arr_false.c	nr	20.02	to	8.78	**0.17**	err	1.07
sum01_bug02_false.c	0.72	1.71	**0.15**	0.29	0.21	6.9	0.83
n.c24_true.c	nr	to	nr	to	to	to	0.6
veris.c_NetBSD-libc__loop_true.c	0.2	1.67	6.48	0.05	0.02	**0.01**	0.26
insertion_sort_false.c	nr	to	to	nr	to	**0.26**	0.63
sum01_false.c	0.77	1.99	**0.12**	0.15	0.38	0.72	0.46
for_infinite_loop_2_true.c	0.2	19.07	nr	nr	**0.01**	0.02	0.29
terminator_02_false.c	nr	1.4	3.25	0.2	**0.01**	0.01	0.42
nec20_false.c	nr	1.39	0.13	0.16	**0.02**	0.12	0.48
verisec_sendmail__tTflag_arr_one_loop_false.c	nr	2.09	to	nr	**0.78**	err	0.39
trex02_true.c	0.2	1.56	nr	nr	**0.01**	0.03	0.37
invert_string_false.c	nr	to	to	**3.65**	err	err	0.6
nec11_false.c	nr	1.56	0.14	0.07	**0.01**	0.02	0.23
bubble_sort_true.c	0.21	19.83	to	to	**0.02**	0.05	3.46
nec40_true.c	0.21	17.24	0.04	0.07	0.02	**0.01**	0.2
veris.c_sendmail__tTflag_arr_one_loop_true.c	0.2	2.19	0.24	nr	**0.01**	0.02	0.44
linear_sea.ch_true.c	nr	nr	nr	**0.18**	to	err	0.32
verisec_NetBSD-libc__loop_false.c	0.43	1.45	4.95	**0.04**	inc	err	0.25
sum01_true.c	nr	to	nr	**0.26**	to	1.24	0.5
sum04_true.c	0.2	1.24	**0.01**	0.08	0.03	0.28	0.31
count_up_down_false.c	0.22	1.41	0.14	0.05	**0.01**	0.02	0.24
trex03_true.c	nr	1.92	nr	nr	**0.03**	0.03	0.82
for_bounded_loop1_false.c	nr	1.44	16.87	0.8	**0.02**	0.09	0.67
n.c40_true.c	0.19	18.01	0.03	0.07	**0.01**	0.02	0.23
heavy_true.c	to	to	to	to	to	to	0.39
lu.cmp_true.c	0.48	1.45	**0.07**	err	24.42	err	0.19
terminator_01_false.c	nr	1.46	0.22	0.11	**0.01**	0.01	0.7
sum03_true.c	0.19	46.46	nr	0.15	**0.06**	0.22	0.38
linear_search_false.c	nr	nr	**0.25**	0.25	1.87	err	0.34
sum04_false.c	0.28	1.46	**0.03**	0.16	0.07	0.71	0.37
eureka_01_false.c	nr	to	to	to	1.34	**0.43**	1.75
sum01_bug02_sum01_bug02_base.case_false.c	0.54	1.7	0.15	0.63	0.12	**0.09**	0.54
while_infinite_loop_4_false.c	0.2	1.43	0.91	0.06	0.01	**0.01**	0.14
eureka_01_true.c	nr	**1.56**	15.39	to	to	7.31	1.15
while_infinite_loop_3_true.c	0.19	1.25	nr	nr	0.01	**0.01**	0.13
vogal_true.c	nr	to	**4.09**	4.19	54.34	to	1.63
count_up_down_true.c	**0.2**	to	nr	nr	to	to	0.27
while_infinite_loop_1_true.c	0.2	1.25	nr	0.03	**0.01**	0.01	0.21
for_infinite_loop_1_true.c	0.2	19.18	nr	nr	**0.01**	0.02	0.22
sum03_false.c	nr	1.52	0.78	0.25	**0.22**	0.69	0.76
invert_string_true.c	nr	7.16	**0.07**	0.96	0.14	inc	1.05
eureka_05_true.c	nr	1.37	**0.08**	0.48	0.87	0.1	0.64

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*