A Program Transformation for Faster Goal-Directed Search

Akash Lal and Shaz Qadeer
Microsoft Research
Email: {akashl, qadeer}@microsoft.com

***Abstract*—**
A goal-directed search attempts to reveal only relevant information needed to establish reachability (or unreachability) of the goal from the initial state of the program. The further apart the goal is from the initial state, the harder it can get to establish what is relevant. This paper addresses this concern in the context of programs with assertions that may be nested deeply inside its call graph—thus, far away interprocedurally from `main`. We present a source-to-source transformation on programs that lifts all assertions in the input program to the entry procedure of the output program, thus, revealing more information about the assertions close to the entry of the program. The transformation is easy to implement and applies to sequential as well as concurrent programs. We empirically validate using multiple goal-directed verifiers that applying this transformation before invoking the verifier results in significant speedups, sometimes up to an order of magnitude.

I. Introduction

Automated program verification attempts to establish reachability (or unreachability) of a goal from the initial state of the program. The goal is usually expressed as the violation of an assert statement in the program. Modern automated program verifiers are typically goal-directed, i.e., they attempt to use program information parsimoniously in order to establish (un)reachability of the goal as efficiently as possible. The challenge of distinguishing relevant from irrelevant and the difficulty of the verification problem increases as the distance of the goal from the initial state becomes larger. This paper addresses this challenge for programs with assertions that may be nested deeply inside its call graph—thus, far away interprocedurally from the program entry point.

Deep assertions are natural in large programs. For instance, in our benchmarks (Section VI), the static nesting depth of assertions (i.e., length of an acyclic path in the call-graph from `main` to a procedure containing an assertion) ranges from 4 to 38 (Fig. 6) and the depth observed on real error traces ranges from 5 to 15 (Fig. 7). At such depths, a naïve strategy of inlining procedures to expose control locations of the assertions is infeasible for analysis because of the exponential cost of inlining.

This paper presents an approach for lifting all assertions to the entry procedure of the program, thus revealing more information about the assertions close to the initial state of the program. Our method is a source-to-source transformation that produces output whose size is a small constant times the size of the input, and applies to both sequential and concurrent programs. We empirically validate using multiple verifiers that applying this transformation before feeding a program to a verifier results in upto order-of-magnitude speedups.

Our transformation is based on the observation that any execution that descends into a call to a procedure `P` either fails inside the call (and doesn't return) or returns from it without failing. We can convert assert statements inside `P` to assume statements if (1) we make a copy of the body of `P`, (2) instrument call sites of `P` to guess whether the call will fail, and (3) either make the call in the *success* case or jump to the copy in the *failure* case. This eliminates the need for making a call in order to reach the control location of the assertion. Further, we only need to make a single copy of the body of `P` regardless of the calling context, because in the *failure* case, control does not need to return to the caller. We also lift assertions outside loops based on the observation that in any execution only the last iteration of the loop (in that execution) can fail. In the presence of concurrency, we exploit the observation that at most one thread can fail.

Contributions. The contributions of this paper are: (1) a novel program transformation that optimizes running time of goal-directed verifiers for programs with deep assertions; and (2) an extensive evaluation over real software that totals over a month of verification time, and shows up to an order of magnitude speedup for two very different verifiers.

Organization. Section II covers background and related work on goal-directed verification techniques. Section III presents an overview of our transformation. Sections IV and V formally present the transformation for a simplified programming language. Section VI presents the evaluation.

II. Background

In order to describe the intuition behind our program transformation, we first discuss some goal-directed verifiers that are based on procedure inlining strategies. We choose these kinds of verifiers for two reason: first, they form a part of our evaluation (Section VI) and second, some inlining strategies have been proposed to specifically address deeply-nested assertions, thus, we compare the effect of our transformation against them.

Bounded model-checking tools (e.g., CBMC [6], [5]) are based on an eager inlining strategy that inlines all procedure calls up to a certain depth to produce a single procedure with all assertions inside it. Eager inlining fails for moderate to large programs because the inlining can result in an exponential explosion, even for small bounds. For instance, in many

ISBN: 978-0-9835678-4-4.

Algorithm 1 Forward and Alternating Inlining Strategies

```
 1: procedure FWD(P)
 2:     if P has an error trace then
 3:         Return BUG
 4:     end if
 5:     P_over := Replace all calls c in P with
        summary(c)
 6:     if P_over has an error trace then
 7:         Let c be a call on that trace
 8:         P := Inline c in P
 9:         Return FWD(P)
10:     end if
11:     Return CORRECT
12: end procedure
```

```
 1: procedure ALT(P)
 2:     if FWD(P) = CORRECT then
 3:         Return CORRECT
 4:     end if
 5:     if P.head = main then
 6:         Return BUG
 7:     end if
 8:     for all callers c of P.head do
 9:         P' := Inline P in c
10:         P'.head := c
11:         if ALT(P') = BUG then
12:             Return BUG
13:         end if
14:     end for
15:     Return CORRECT
16: end procedure
```

of the benchmarks used in this paper, eager inlining ran out of memory even before the analysis was started.

To avoid the cost of eager inlining, there are several proposed *lazy* inlining strategies that inline procedure on-demand and in a goal-directed manner. Techniques such as *structural abstraction* [2], *inertial refinement* [21], and *stratified inlining* [14] are all forward-inlining strategies, described abstractly by the method FWD of Alg. 1.

Forward Inlining. FWD takes a partially-inlined program P as input. (One can think of P as a single procedure containing some procedure calls.) Initially, P is just the body of `main`. FWD checks if P contains a bug without going through a procedure call (line 2). If not, then it picks a *relevant* procedure call made by P (line 7), inlines the body of the callee (line 8) and repeats. The choice of picking relevant calls is guided using procedure summaries (that are either pre-computed or inferred on the fly): if no error trace in P_{over} goes through a call c then this proves that no error trace of the original program goes though c. A *default* summary based on mod-set information, i.e., a procedure can arbitrarily modify variables that it can touch, can always be used. FWD, even with default summaries, has been shown to be much better than eager inlining in some contexts [2], [14]. Further, one can treat loops as tail-recursive procedures to extend FWD to perform loop unrolling as well.

FWD raises two technical concerns: first, what do procedure summaries mean in the presence of assertions, and second, what does it mean to query P_{over} for error when it may not even contain an assertion? Both these question are answered using an *error-bit instrumentation*. As pre-processing, we add a Boolean global variable `err` to the program; it is set to *true* if and only if an assertion fails; and all procedures immediately return when `err` is *true*. Then procedure summaries can use `err` to distinguish failing executions from non-failing ones. Moreover, we simply query P_{over} for a trace that ends with `err` set. We note that the error-bit instrumentation results in a program with the only assertion in `main`. However, it does not reveal any information about the original assertions themselves.

We illustrate FWD using the example in Fig. 1. This program has two global variables `s` and `g`. The entry procedure `main` initializes `s` and `g` and calls `P1`. The procedure `P1` is the first

```
var s, g: int;
procedure main()
{  s := 0; g := 1;
   P1();
}
procedure P1()
{ P2(); P2(); }
...
```

```
procedure Pn()
{ while (*) {
   if (g == 1)
     Open();
   Close();
  }
}
```

```
procedure Open()
{ s := 1; }

procedure Close()
{ assert s > 0;
  s := 0;
}
```

Fig. 1: An example program

in a chain of procedures `P1`,...,`Pn` each of which (except the last) calls its successor twice. `Pn` contains a nondeterministic loop that calls `Open` and `Close` in alternation. The assert statement inside `Close` cannot fail.

Suppose we wish to explore all behaviors of this program up to R loop iterations. In this case FWD will inline $O(2^n)*O(R)$ procedures to conclude unreachability (under R) when using default summaries because no call will be deemed irrelevant. This number comes down to $O(1)$ when FWD has the following (inductive) procedure summaries available for each `Pi`: `(old(g) == 1 && old(s) == 0) ==> (s == 0 && !err)`, where `old(v)` refers to the value of `v` at the beginning of the procedure. This says that if `g` and `s` are `1` and `0`, respectively, at the beginning of `Pi` then when `Pi` returns, the value of `s` is still `0` and `err` has not been set. Clearly, given this summary for `P1`, FWD can conclude the absence of assertion failure just looking at `main`.

Alternating Inlining. Other inlining strategies include both backward and forward search [1, Section 4.2] [22], captured abstractly using ALT in Alg. 1. It starts with P as a procedure with an assertion. It conducts a forward search (line 2) to find an error trace from the initial state of P. If such a trace is found, it picks a caller of P, inlines P inside it and repeats until the search reaches `main`. An interesting remark is that ALT does not require the error bit instrumentation. This is because it starts with the assertion that it wishes to violate, and all procedures inlined during the call to FWD are constrained to not fail. Thus, all summary computation can be done assuming fail-free executions.

On Fig. 1, ALT will inline $O(2^n) * O(R)$ procedures when using default summaries. However, using just the (inductive) fact that `g == 1` is a valid precondition of each `Pi`, this number comes down to $O(1)$. This is because when the search is at procedure `Pn`, then under this precondition, ALT can already prove the absence of assertion violations (line 3) without enumerating the calling contexts of `Pn`.

Thus, different inlining strategies can involve different amount of inlining, and put different amount of stress on invariant and summary generation.

III. OVERVIEW OF OUR PROGRAM TRANSFORMATION

In this section, we informally describe our novel contribution, a semantics-preserving source-to-source transformation that lifts all assert statements in a program into its entry procedure. As explained in Section I, our transformation is based on the simple observation that any execution that descends into a call to a procedure `P` either fails inside the call or returns from it. We will convert all assert statements

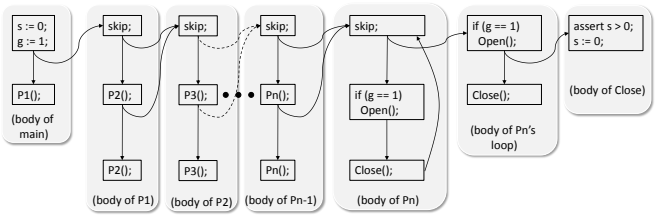

Fig. 2: Control-flow graph of transformed procedure `main`

$$
\begin{array}{llll}
P & \in & Prog & ::= (gs, ps) \\
p & \in & Proc & ::= (x, is, os, vs, st) \\
vs & \in & Vars & ::= \cdot \mid x : t, vs \qquad gs, is, os \in Vars \\
ps & \in & Procs & ::= \cdot \mid p, ps \\
st & \in & Stmt & ::= l\!:\!assume\ e \mid l\!:\!assert\ e \mid l\!:\!xs\ :=\ es \mid \\
 & & & \quad l\!:\!havoc\ xs \mid l\!:\!goto\ ls \mid l\!:\!loop\ st \mid \\
 & & & \quad l\!:\!call\ xs\ :=\ x(es) \mid l\!:\!async\ x(es) \mid l\!:\!yield \mid \\
 & & & \quad st; st \\
xs & \in & Names & ::= \cdot \mid x, xs \qquad x \in Name \\
es & \in & Exprs & ::= \cdot \mid e, es \qquad e \in Expr \\
ls & \in & Labels & ::= \cdot \mid l, ls \qquad l \in Label \\
t & \in & Type &
\end{array}
$$

Fig. 3: Program syntax

inside `P` to assume statements and simulate failures in the body of `P` by nondeterministically jumping to a copy of the body of `P` at a call site.

Fig. 2 shows, as a control-flow graph, the result of our transformation on the `main` procedure of our running example from Fig. 1. The bodies of all other procedures remain the same except for `Close` in which `assert s > 0` is converted to `assume s > 0`. The execution of transformed `main` begins in the top-left block with the initialization of the global variables. Next, it can non-deterministically choose to call `P1` or jump to a copy of the body of `P1`. The two calls to `P2` in the body of `P1` are similarly instrumented, and so on.

The instrumentation of the body of `Pn` is interesting because it contains a loop. In addition to lifting assertions out of procedure calls, we would also like to lift them out of loops. Our insight is that it suffices to allow only the final iteration of the loop to fail. Therefore, we can make a copy of the loop body, convert assert statements inside the loop to assume statements, and then nondeterministically execute the copy of the body after the loop at most once.

It is worth noting that in Fig. 2, we did not make a copy of the body of procedure `Open`. We could do this optimization because it was possible to statically determine that a call to `Open` cannot fail.

When FWD is applied to the transformed program, it only inlines $O(1)$ number of procedures to conclude CORRECT. (In particular, it only needs to inline the call to `Open` from the new `main`.) The reason is that the value flow between the initialization of `g` and the conditional expression guarding the call to `Open` is apparent at the top-level without any intervening loops and calls, even under default summaries. In this case, inlining the call to `Open` is sufficient to discharge the assertion. Thus, no summary or invariant generation was required for this example after our transformation. This example provides intuition for the speedup on programs with an unreachable goal, however, pruning infeasible paths also translates to finding the goal faster when reachable. This is confirmed by our experiments.

While we have chosen to evaluate our program transformation against lazy inlining strategies (as each address the issue of deep assertions), our approach is more general. It is not tied to a particular analysis. It simply produces a new program that can be fed to any verifier, with the hope of speeding up the verifier. For instance, our evaluation uses the YOGI verifier for C programs that is based on predicate-abstraction and doesn't directly implement an inlining strategy. This point is further emphasized when dealing with concurrent programs, as we are not aware of inlining strategies that directly apply to concurrent programs.

Remark: Here we note that our approach is inspired by "Phase 2" of the RHS algorithm [19], [20]. RHS is the standard tabulation-based algorithm for interprocedural dataflow analysis. It works in two phases: the first phase computes procedure summaries bottom-up in the call graph. The second phase replaces procedure calls with the summaries and deletes return edges. This transformation is similar to ours, however, we do not use summaries and our target is goal-directed program verification, not dataflow analysis. Moreover, our transformation has special handling for loops and concurrency.

IV. A SIMPLE PROGRAMMING LANGUAGE

We present a core programming language, similar to Boogie [3], for formalizing our program transformation. The syntax of the language is presented in Fig. 3. A program P is a tuple comprising a set of global variable declarations gs and a set of procedure declarations ps that is assumed to contained a distinguished procedure called `main`. Each procedure is a tuple comprising its name x, input parameters is, output parameters os, local variables vs, and a statement st. As notation, for a procedure $f = (x, is, os, vs, st)$, let $\mathit{name}(f) = x$, $\mathit{input}(f) = is$, $\mathit{output}(f) = os$, $\mathit{locals}(f) = vs$, and $\mathit{code}(f) = st$. We assume, without loss of generality, that `main` is never called and it does not have output variables.

A statement st is a ";"-separated list of a label l and one of the following—*assert*, *assume*, *assignment*, *havoc*, *goto*, *loop*, *call*, *async*, or *yield*. In our presentation, we ignore the syntax of expressions and types and assume the existence of a type checker for validating that the program is well-formed. Further, we may sometimes omit writing the label of a statement, in which case it is assumed to have a fresh label that is not used elsewhere in the program. Statement labels must be unique and cannot be re-used.

The control flow in our language is straightforward. The statement $goto\ ls$ causes control to non-deterministically jump to some label in ls; the type checker ensures that the labels exist in the same procedure or enclosing loop. For all other statements, control implicitly moves to the next statement by following the sequential composition (";") operator. If there is no next statement, then execution of the statement terminates.

$assert\ e$ fails if e evaluates to false in the current state and otherwise leaves state unchanged. $assume\ e$ blocks if e

$[\![l: assume\ e]\!]_{\text{stmt}} = l: assume\ e$
$[\![l: assert\ e]\!]_{\text{stmt}} = l: assert\ e$
$[\![l: xs\ :=\ es]\!]_{\text{stmt}} = l: xs\ :=\ es$
$[\![l: havoc\ x]\!]_{\text{stmt}} = l: havoc\ x$
$[\![l: goto\ ls]\!]_{\text{stmt}} = l: goto\ ls$
$[\![st_1; st_2]\!]_{\text{stmt}} = [\![st_1]\!]_{\text{stmt}}; [\![st_2]\!]_{\text{stmt}}$
$[\![l: call\ xs\ :=\ x(es)]\!]_{\text{stmt}} =$
$l: if(\star) then\{call\ xs\ :=\ x(es)\}$
$else\{input(x)\ :=\ es; havoc\ locals(x); goto\ x^{entry}\}$
$[\![l: loop\ st]\!]_{\text{stmt}} = l: loop\ \overline{st}; if(\star) then\{[\![st]\!]_{\text{stmt}}\} else\{skip\}$
$[\![(x, is, os, vs, st)]\!]_{\text{proc}} = (x, is, os, vs, \overline{st})$
$[\![(p, ps)]\!]_{\text{proc}} = ([\![p]\!]_{\text{proc}}, [\![ps]\!]_{\text{proc}})$
$[\![(gs, (\texttt{main}, is, \cdot, vs, st), ps)]\!]_{\text{prog}} = (gs,$
$(\texttt{main}, is, \cdot, vs \cup \bigcup_{p \in ps} input(p) \cup output(p) \cup locals(p),$
$[\![st; die; x_1^{entry}: skip; code(x_1); die; \cdots; x_n^{entry}: skip; code(x_n); die]\!]_{\text{stmt}}),$
$[\![ps]\!]_{\text{proc}})$ where $ps = (p_1, \cdots, p_n)$ and $name(p_i) = x_i$

Fig. 4: Transforming sequential programs

evaluates to false in the current state and otherwise leaves state unchanged. $xs\ :=\ es$ is a parallel assignment that evaluates es in the current state and updates variables xs to the result. $havoc\ xs$ puts nondeterministically chosen values into each variable in xs. $loop\ st$ is a nondeterministic structured loop and executes st zero or more times. $call\ xs\ :=\ x(es)$ is call to procedure x with inputs es; the output of the procedure call is received in variables xs. $async\ x(es)$ is an asynchronous call to procedure x with inputs es; the call is executed in a new thread that executes concurrently with all existing threads. The multithreading model in our language is cooperative and nondeterministic; $yield$ yields control to a nondeterministically chosen thread.

For convenience, we also use a statement $if(\star) then\{st_1\} else\{st_2\}$ that denotes non-deterministic branching between two statements. We use it as syntactic sugar over using $goto$ statements.

V. Program transformation

We begin by presenting our transformation for sequential programs in Fig. 4 and generalize it to concurrent programs in Fig. 5. We use *skip* and *die* to compactly denote $assume\ true$ and $assume\ false$, respectively. Our transformation depends on an initial renaming of variables and labels in the program to make them globally distinct. This initial renaming is standard and we do not present it here. Further, for a statement st, let $\overline{st}$ be the same statement where all occurrences of $assert\ e$ in st are converted to $assume\ e$.

A. Transforming sequential programs

Fig. 4 describes three transformations: $[\![.]\!]_{\text{stmt}}$ for statements, $[\![.]\!]_{\text{proc}}$ for procedures and $[\![.]\!]_{\text{prog}}$ for programs. First, note that the transformation of a procedure simply disables all assertions in the procedure. The transformation on a program leaves the set of global variables unchanged and disables assertions in all procedures except `main`. The `main` procedure is transformed by absorbing the bodies of all other procedures (along with their input, output and local variables) and applying the statement transformer on them. It is easy to show that $[\![P]\!]_{\text{prog}}$ can only have assertions in `main`.

$[\![l: yield]\!]_{\text{stmt}} = l: yield$
$[\![l: async\ x(es)]\!]_{\text{stmt}} = l: if(\star) then\{async\ x(es)\}$
$else\{assume\ flag = nil; a_{input(x)}\ :=\ es; flag\ :=\ c_x\}$
$[\![(gs, ps)]\!]_{\text{prog}} = (gs \cup \{flag\} \bigcup_{p \in ps} a_{input(p)},$
$(\texttt{newmain}, is, \cdot, vs \bigcup_{p \in ps} input(p) \cup output(p) \cup locals(p), [\![st]\!]_{\text{stmt}}), [\![ps]\!]_{\text{proc}})$
where $ps = (\texttt{main}, p_1, \cdots, p_n)$, $name(p_i) = x_i$ and
$st \stackrel{\text{def}}{=} flag\ :=\ nil;$
$if(\star) then\{flag\ :=\ c_{\texttt{main}}; goto\ \texttt{main}^{entry}; die\} else\{skip\};$
$async\ main(is); yield; goto\ l_{x_1}, \cdots, l_{x_n}; die;$
$l_{x_1}: assume\ flag = c_{x_1}; input(x_1)\ :=\ a_{input(x_1)}; goto\ x_1^{entry}; die;$
$\cdots$
$l_{x_n}: assume\ flag = c_{x_n}; input(x_n)\ :=\ a_{input(x_n)}; goto\ x_n^{entry}; die;$
$[\![\texttt{main}^{entry}: skip; code(\texttt{main}); die]\!]_{\text{stmt}};$
$[\![x_1^{entry}: skip; code(x_1); die; \cdots; x_n^{entry}: skip; code(x_n); die]\!]_{\text{stmt}}$

Fig. 5: Transforming concurrent programs

Let us now look at the statement transformer $[\![.]\!]_{\text{stmt}}$. It is non-trivial only for procedure calls and loops. It transforms a procedure call of x to a non-deterministic branch. The *then* branch simulates an execution where the procedure call succeeds. In this case, the call is left untouched. However, note that x does not have assertions in the transformed program, thus a call to it cannot fail. The *else* branch simulates an execution where the procedure call fails. In this case, we simply jump to x^{entry} where a copy of the body of x resides. Note the use of die in the $[\![.]\!]_{\text{prog}}$ transformation. This prevents the execution of, say, x_2's body to fall through onto the body of x_3. Thus, a jump to the body of a procedure cannot ever return (but it may fail).

The statement transformation for loops works by first peeling off the last iteration of the loop. ($loop\ st$ is equivalent to $loop\ st; if(\star) then\{st\} else\{skip\}$.) Next, the new loop's body is not allowed to fail ($\overline{st}$), because only the last iteration of a loop can fail. The statement transformer is applied recursively to the last iteration.

B. Transforming concurrent programs

The transformation described in the previous section, although adequate for lifting all assertions to the entry procedures of all threads, is inadequate for lifting all assertions to just the `main` block of the initial thread. This section extends the transformation described earlier to achieve this goal.

Fig. 5 defines the statement transformer for *yield* and *async* procedure calls. It also redefines the program transformation. The rest is borrowed over from Fig. 4. The main insight behind these transformations is that any erroneous execution has exactly one assertion failure which stops the execution. Therefore, it suffices to allow at most one thread, either initial or dynamically-created, to fail. The start procedure of a dynamically-created thread is one of a finite number of procedures that are targets of asynchronous procedure calls. We introduce fresh constants including the special constant nil and a constant c_x for each procedure in the input program with name x; these constants are assumed to be distinct from each other. We also introduce a fresh global variable *flag* whose value is one of these freshly introduced constants; this variable is initialized to nil. During the execution of the transformed

program its value changes at most once from nil to some constant c_x. The final value of *flag*, if different from nil, represents the entry procedure of the potentially failing thread.

The transformation of an asynchronous call $async\ x(es)$ is a non-deterministic choice. One choice is to keep the asynchronous call, but to a procedure that cannot fail (recall the procedure transformation from Fig. 4). The other choice is to atomically update *flag* from nil to c_x, which simulates the creation of a failing instance of x. (The failing instance executes in the entry procedure of the transformed program, discussed later in program transformation rule.) Blocking on the condition $flag = nil$ ensures that at most one failing instance is created. We use additional global variables a_v (where v is an input argument to some procedure) for storing the arguments of the failing thread instance.

The $[\![.]\!]_{\text{prog}}$ transformation is more sophisticated. It works by creating a new procedure, called `newmain` that is understood to be the entry procedure of transformed program. It consists of the bodies of all other procedures, including `main`. It starts by initializing *flag* to nil. Next, it decides if the `main` thread is the one that fails; if so, it jumps to `main`. Otherwise, it spawns `main` as a separate thread (which cannot fail) and non-deterministically jumps to a location l_x for some procedure x. The location l_x waits for *flag* to be set to c_x, grabs input arguments from a_v variables, and jumps to the body of x.

C. Correctness

Let P be a program where all variables and labels are globally distinct. The most important property of our transformation is that it is failure-preserving. Therefore, verifying the original program is equivalent to verifying the transformed program.

Theorem 5.1: P fails an assertion if and only if $[\![P]\!]_{\text{prog}}$ fails an assertion.

The following theorem states that we succeeded in our objective of lifting all assert statements out of loops and procedures.

Theorem 5.2: In $[\![P]\!]_{\text{prog}}$, no procedure other than the entry procedure can have assertions. Further, even loop statements in the entry procedure cannot have assertions.

Next, we state a property about the compactness of our transformation. Let $|P|$ denote the size of the program P. The *loop nesting depth* of a program is defined recursively as follows.

$$\begin{aligned} LND(ps) &= max(\{LND(p) \mid p \in ps)\}) \\ LND((x, is, os, vs, st)) &= LND(st) \\ LND(st_1; st_2) &= max(LND(st_1), LND(st_2)) \\ LND(l\!:\!loop\ st) &= LND(st) + 1 \\ LND(\_) &= 1 \end{aligned}$$

Theorem 5.3: $|[\![P]\!]_{\text{prog}}| = |P| \times (LND(P) + c)$ for a small constant c.

Finally, our transformation enjoys the desirable property that if the input program is recursion-free and has only structured loops, then so is the output program.

Theorem 5.4: If P is recursion-free and each procedure has an acyclic control-flow graph then $[\![P]\!]_{\text{prog}}$ is recursion-free and each procedure has an acyclic control-flow graph.

VI. Evaluation

We refer to the transformation of Section V as the *deep-assert* (DA) instrumentation. We conducted extensive experiments to evaluate its effect on the running time of two different verifiers:

1) Corral [14] is an SMT-based verifier that accepts Boogie programs [16] as input. It consists of an outer loop of abstraction refinement. Inside the loop, it verifies a program using either Fwd or Alt of Alg. 1, based on stratified inlining [14] and alternating inlining [22], respectively.
2) Yogi [4], [10] is a verifier for C programs. It alternates between test generation (for proving "reachability" information) and automated predicate abstraction (for proving "unreachability" information).

We chose Yogi because: first, it currently uses the error-bit instrumentation (Section II). Second, Yogi has been highly optimized over several years of research and development [18], [11], [4], [10], thus, any performance improvement is considered significant. Third, it is a "third-party" tool; we were never a part of the design or implementation of Yogi.

Let SI and AT refer to Corral with stratified inlining and alternating inlining, respectively, and let SI+DA refer to applying our deep-assert transformation followed by running Corral with stratified inlining. Note that once the deep-assert transformation is executed then using SI or AT is identical as all assertions would be in `main`.

Corral uses Houdini [9] for generating program invariants and procedure (and loop) summaries. Let SI+H, AT+H and SI+DA+H refer to configurations when Houdini is enabled. Houdini requires invariant templates to be supplied by the user. Invariant generation in Yogi is fully automated.

Corral and Yogi use different IR representation for programs. The implementation of the deep-assert instrumentation for Corral was 969 lines of C# code[1] and for Yogi was 166 lines of OCaml code.

All experiments were conducted on a server class machine with two Intel(R) Xeon(R) processors (16 logical cores) executing at 2.4 GHz with 32 GB RAM. Different verification instances were executed in parallel, with at most 16 instances (one per core) executing in parallel at any given time.

Static Driver Verifier. Our first set of experiments is using the Static Driver Verifier (SDV) [17]. SDV is a commercial-grade tool offered by Microsoft to third-party driver developers. We collected a set of real device drivers that have been historically challenging for SDV, shown in Fig. 6. The drivers total 115KLOC, and additionally link against libraries of size 75KLOC. Fig. 6 also gives the number of procedures (#Procs) and "Assert Depth", which is a pair consisting of the smallest and largest acyclic path in the call graph from the entry point to a procedure containing an assert. This is a static measure for how deep the assertions were in the program. The last column lists the number of verification instances for

[1] available open source at corral.codeplex.com.

Name	KLOC	#Procs	Assert Depth (min,max)	#Verif. Instances
fdc_fail	9.2	216	4-18	226
kbdclass	7.1	230	4-31	252
daytona	21.5	345	4-27	316
parport	33.9	531	4-21	169
sys	2.2	108	4-27	596
isapnp	14.1	286	4-18	94
mouser	7.4	190	4-38	600
modem	14.4	289	4-26	157
kerneldriver	5.0	183	4-34	106
Total	115+75	2378	4-38	2516

Fig. 6: Details of SDV benchmarks

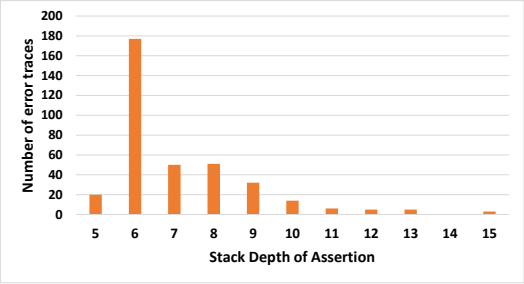

Fig. 7: Stack depth of SDV error traces

a driver: SDV verifies multiple properties of a driver and in doing so, generates multiple different verification instances. For our purpose, a verification instance is simply a program with assertions. SDV generated verification instances have no recursion (usually drivers don't have recursion, and even when they do, SDV statically unrolls the recursion up to a small bound). Moreover, all loops are structured (i.e., the control-flow graph of procedures are *reducible*), in which case we can compile loops to use our *loop* statement. Fig. 7 shows the stack depth at which the failing assert was reached among all the error traces found in the benchmark suite. It shows a reasonable range and variation.

SDV generates a set of predicates $R_1, \cdots, R_n, S_1, \cdots, S_m$ for each verification instance, based on the property that it is checking [14]. The R_is are predicates over the input state of a procedure; they serve as templates for preconditions. The S_is are templates for postconditions (summaries). SI uses the error-bit instrumentation; let `err` be the error bit summarizing if an assertion has failed or not (see Section II). SI+H uses HOUDINI to look for procedure summaries of the following form: `!err ==>` S_i (i.e., S_i is a summary when the procedure doesn't fail) and R_j `==> !err` (i.e., under R_i, the procedure doesn't fail). AT+H doesn't use the error-bit instrumentation; it looks for summaries of the form S_i and preconditions of the form R_j. While summaries can be inferred bottom-up in the call graph, inferring preconditions requires a top-down pass as well. SI+DA+H also doesn't use the error-bit instrumentation (there is no need because all assertions are lifted to `main` by DA). Further, it only looks for summaries of the form S_i; the templates R_j are dropped as our deep-assert transformation reduces the need for preconditions.

Aggregate results across all verification instances are shown in Fig. 11. The table lists the total number of instances that

Algorithm	#TO	#Bnd	#Bugs	#Proof	Houd. (1000 s)	Time (1000 s) Bug	Time (1000 s) No-bug
SI	510	477	348	1181	0	23	154
AT	314	638	345	1219	0	31	126
SI+DA	**213**	**383**	**363**	**1557**	0	**21**	**93**
SI+H	73	129	360	1954	76	35	156
AT+H	126	226	350	1814	115	47	205
SI+DA+H	**43**	**127**	**363**	**1983**	**53**	**32**	**123**

Fig. 11: Results, in aggregate, for the SDV benchmarks

timed out after 2000 seconds (#TO), hit the search bound (i.e., inconclusive) (#Bnd), produced an error trace (#Bugs), or proved the instance correct (#Proof). The other columns list the total time taken by HOUDINI (Houd), and the time spent by CORRAL (inclusive of time spent by HOUDINI) on buggy and non-buggy instances. Non-buggy instances include both bound-hit and proofs, but not timeouts. Times are reported in units of 1000 seconds. The entire table took 41 days of verification time.

The table shows advantages of the deep-assert instrumentation along several dimensions. SI+DA and SI+DA+H have much fewer timeouts, find more bugs, prove more instances correct, and take the least amount of time. Using HOUDINI significantly reduces the number of timeouts and increases the number of instances proved correct (for each of SI, AT, and SI+DA). These numbers suggest that the templates used by HOUDINI were complete to a good extent. However, the time taken by HOUDINI is a significant fraction of the total running time. Thus, optimizing HOUDINI usage is important. The table shows that the simplification of templates provided by DA improves the running time of HOUDINI. Because ALT requires preconditions for pruning, AT+H spends the maximum amount of time in HOUDINI—more than twice as much as SI+DA+H. Consequently, AT+H is the slowest among other configurations with HOUDINI. This indicates that ALT imposes a stricter demand for invariants for pruning search. SI+DA, on the other hand, does well even without invariant generation; in fact, it finds all the 363 bugs without the help of HOUDINI.

Fig. 8 presents a more detailed comparison of the running times of SI and SI+DA. The scatter plot (on the left) is the distribution of running times: each dot is a single verification instance. The chart on the right summarizes the number of instances in which DA resulted in a particular speedup (computed as a fraction of the running time). "Infinity" means that a timeout was eliminated, and "-Infinity" means that a timeout was introduced. For example, there are 54 instances in which SI+DA is at least 10 times faster than SI. The numbers on top of the bars indicate the average running time of SI (in seconds) on an instance that falls in that bar. For example, whenever SI+DA was 5 to 10 times faster than SI, the average time taken by SI was 434.2 seconds. These numbers show that the speedup was obtained on non-trivial instances. Further, only 6 timeouts were introduced, and 303 were eliminated by DA. Only 5 instances experienced a slowdown worse than a factor of 2 (see the bar "< 0.5"). There are 1726 instances with speedup in the range 0.5 to 1.75. These are not shown in the figure, moreover, their average running time was just 69

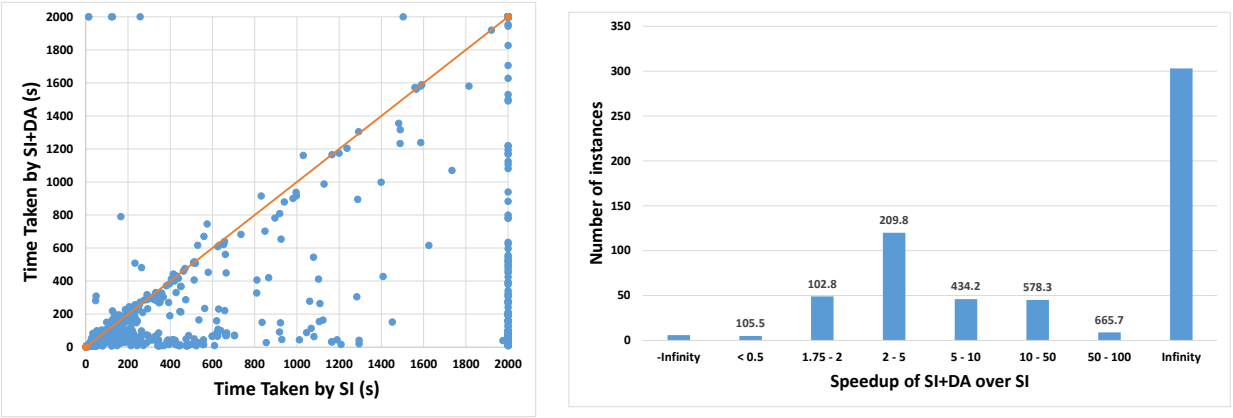

Fig. 8: Comparisons of running time between SI and SI+DA

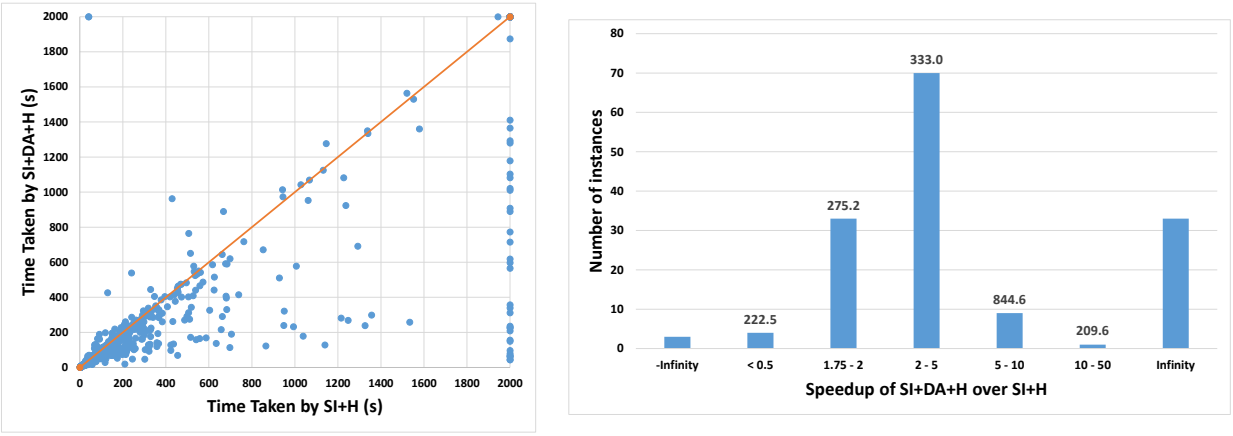

Fig. 9: Comparisons of running time between SI+H and SI+DA+H

seconds. One can also visually observe high density of dots near the origin of the scatter plot.

Fig. 9 shows similar graphs for SI+DA+H against SI+H. In this case, 33 timeouts were eliminated and only 3 introduced by DA. Only 4 instances observed a slowdown worse than a factor of 2. There are 2323 instances with speedup in the range 0.5 to 1.75 with an average running time of 76 seconds.

Fig. 10 shows the effect of DA on the running time of YOGI. The overall speedup is a modest 9% but this increases to as much as 50% (i.e, a factor of 2 faster) on harder instances that take at least 600 seconds. The benchmarks used for YOGI were the same set of drivers as mentioned in Fig. 6, but for a subset of the verification instances (total 802). Because YOGI does not support features like bitvector reasoning and arrays, we disabled some of the SDV properties when using YOGI. DA eliminated 8 timeouts and only 1 was introduced. As before, the slowdowns are mostly on trivial instances. The average running time on such instances was less than 2 minutes. The harder instances, with longer running time, usually show a speedup.

The scatter plot of Fig. 10 shows a greater spread than for CORRAL (Figs. 8 and 9). We believe this is because CORRAL uses a more powerful (SMT-based) intraprocedural analysis

Program	LOC	Assert Depth	#Inst-ances	CORRAL (sec)	CORRAL+DA (sec)
daytona	488	3-5	5	460.6	**407.4**
kbdclass	694	3-4	2	713.9	**641.7**
mouclass	581	3-4	7	3877.8	**2964.0**
ndisprot	592	3-5	3	**314.9**	345.9
pcidrv	449	3-5	6	**796.4**	988.5
total	2804	3-5	23	6163.9	**5347.7**

Fig. 12: Results on concurrency benchmarks

and this matches well with the programs produced by DA as they have a large `main` procedure.

Memory Consumption: For SDV benchmarks, we observed that the ratio of $|[\![P]\!]_{\text{prog}}|$ to $|P|$ ranged from 1.1 to 1.6, which is much smaller than the worst-case mentioned in Thm. 5.3. This is because bodies of nested loops tend to be very small compared to the rest of the program. (DA copies the body of a loop as many times as its nesting depth.) Moreover, many procedures cannot statically reach an assert, thus they need not be copied into `main` by the instrumentation. Despite the increase in program size, DA still reduces memory consumption because of the decreased analysis complexity. On average, the peak memory usage of SI+H was 461MB, for AT+H it was 663MB, and for SI+DA+H it was 443MB.

ISBN: 978-0-9835678-4-4.

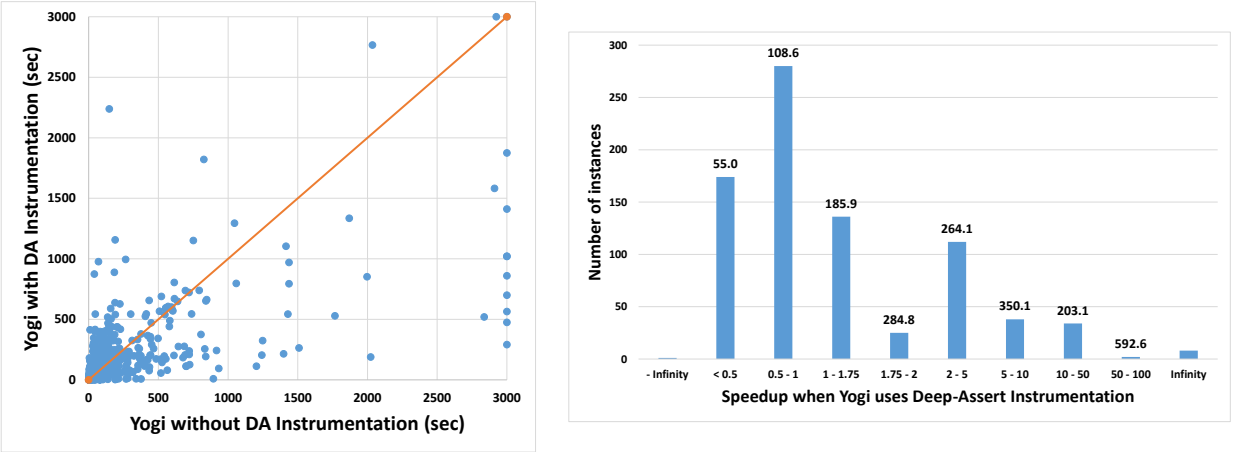

Fig. 10: Yogi with and without deep-assert instrumentation

Concurrency: One scalable approach for the analysis of large (multiple-procedure) concurrent programs is the process of *sequentialization* [13], [7], [8], [15] where a concurrent program is transformed to a sequential program and then verified using a sequential analysis tool. CORRAL supports such a sequentialization; it feeds the resulting sequential program to stratified inlining.

Remark. Sequentializations only preserve end-state reachability and require a variant of the error-bit instrumentation for assertions.[2] This implies that the generated sequential programs have an assertion only at the end of `main`. Consequently, any transformation for revealing information about deep assertions needs to be done on the concurrent program before the sequentialization, as our transformation does.

Fig. 12 reports results on concurrent programs (obtained from [13]) using CORRAL. The improvement is a modest 13% overall, and the assert depth of the benchmarks is also quite shallow. We leave further investigation on concurrent benchmarks for future work.

Summary: We note that there are several other choices of verifiers and it is possible that our program transformation may interact differently with the search heuristics of the verifier. However, our experimental evaluation shows a large potential for speedups, especially given that we do not algorithmically modify the verifier. Further, the program transformation can be applied with any verifier, and takes relatively minimal effort to implement (a few hundred lines of code).

REFERENCES

[1] D. Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[2] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification*, 2007.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, pages 364–387, 2005.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[5] CBMC: Bounded Model Checking for ANSI-C. http://www.cprover.org/cbmc/.

[6] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[7] M. Emmi, A. Lal, and S. Qadeer. Asynchronous programs with prioritized task-buffers. In *Foundations of Software Engineering*, 2012.

[8] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *Principles of Programming Languages*, 2011.

[9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, pages 500–517, 2001.

[10] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Principles of Programming Languages*, pages 43–56, 2010.

[11] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Foundations of Software Engineering*, 2006.

[12] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *Computer Aided Verification*, 2009.

[13] S. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification*, 2009.

[14] A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *Computer Aided Verification*, 2012.

[15] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1), 2009.

[16] K. R. M. Leino. Boogie: An intermediate verification language. http://research.microsoft.com/en-us/projects/boogie.

[17] Microsoft. Static driver verifier. http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx.

[18] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *International Conference on Software Engineering*, pages 355–364, 2010.

[19] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages*, 1995.

[20] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[21] N. Sinha. Modular bug detection with inertial refinement. In *Formal Methods in Computer Aided Design*, 2010.

[22] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *Computer Aided Verification*, pages 599–615, 2012.

[2]There are sequentializations that preserve assertions [12], but these have not yet been shown to be scalable to programs in real languages like C.