Parallel Bug-Finding in Concurrent Programs via Reduced Interleaving Instances

Truc L. Nguyen*, Peter Schrammel†, Bernd Fischer‡, Salvatore La Torre§, Gennaro Parlato*
* University of Southampton, UK, trucnguyenlam@gmail.com, gennaro@ecs.soton.ac.uk
† University of Sussex, UK, p.schrammel@sussex.ac.uk
‡ Stellenbosch University, South Africa, bfischer@cs.sun.ac.za
§ Università degli Studi di Salerno, Italy, slatorre@unisa.it

***Abstract*—Concurrency poses a major challenge for program verification, but it can also offer an opportunity to scale when subproblems can be analysed in parallel. We exploit this opportunity here and use a parametrizable code-to-code translation to generate a set of simpler program instances, each capturing a reduced set of the original program's interleavings. These instances can then be checked independently in parallel. Our approach does not depend on the tool that is chosen for the final analysis, is compatible with weak memory models, and amplifies the effectiveness of existing tools, making them find bugs faster and with fewer resources. We use Lazy-CSeq as an off-the-shelf final verifier to demonstrate that our approach is able, already with a small number of cores, to find bugs in the hardest known concurrency benchmarks in a matter of minutes, whereas other dynamic and static tools fail to do so in hours.**

***Index Terms*—Verification, concurrency, sequentialization, swarm verification**

I. Introduction

Processor development has reached the point where clock speeds can no longer be increased easily, but processors contain multiple cores that work in parallel. Therefore, to achieve higher performance software must necessarily be concurrent.

Unfortunately, developing correct, scalable, and efficient concurrent programs is a complex and difficult task, due to the large number of possible concurrent executions that must be considered. Modern multi-core processors and weak memory models (WMMs) make this task even harder, as they introduce additional executions that confound the developers' reasoning. Due to these complex interactions, concurrent programs often contain bugs that are difficult to find, reproduce, and fix.

Existing automatic bug-finding techniques and tools are not effective when facing concurrent programs. They struggle particularly with programs that contain rare concurrency bugs, i.e., programs where only a few, specific interleavings violate the specification. For techniques that analyze executions explicitly, finding rare bugs is like looking for a needle in a haystack. For techniques that analyze all executions collectively using symbolic representations, finding rare bugs is also challenging due to the large amount of memory required for the analysis. As a result, we currently do not have techniques and tools that can reliably find such rare bugs.

Partially supported by EPSRC grants no. EP/M008991/1 and no. EP/P022413/1, and MIUR-FARB 2014-2017 grants.

 Record was published in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017. IEEE Computer Society 2017, ISBN 978-1-5386-2684-9. http://dx.doi.org/10.1109/ASE.2017.8115686

Although concurrency is clearly a problem for reasoning about programs, it also offers a chance to scale up verification, as suggested by Holzmann et al. [1]: *"... to scale applications of logic model checking to larger problem sizes then, we must be able to leverage the availability of potentially large numbers of processors that run at [...] relatively low speed."*

Different approaches have been tried out to achieve this leveraging, with varying degrees of success. In (truly) *distributed algorithms*, multiple processors are running the same algorithm jointly on the same problem and periodically exchange information. However, verification using distributed model checking techniques (e.g. [2], [3]) has had limited success because they need to share too much information, leading to high communication overheads and contention.

In *strategy competition*, multiple processors are running different variants of the same underlying algorithm independently (i.e., without exchanging information) on the same problem; the first variant that produces a definitive answer (i.e., counterexample or proof) "wins" and aborts the others. This exploits the fact that complex search procedures such as model checkers [4], [1], SAT solvers [5], or first-order theorem provers [6], [7], [8] have many control parameters and strategies that can be used to explore different parts of the search space. Holzmann et al. have applied this idea in *swarm verification* [4], [1] to scale up model checking based on explicit state space exploration. More specifically, their approach is to spawn a large number of instances (the "swarm") of the SPIN model checker, each with different parameters and search strategies; each instance runs an incomplete search, but in aggregate the swarm substantially outperforms an exhaustive search.

In this paper, we propose a different approach called *task competition*: we run the *same* algorithm on multiple processors, again in competition without information exchange, but now on *different* and *easier* verification tasks derived from the original problem. Specifically, each task captures a subset of the program's interleavings under analysis, in a way that each of such interleavings is captured by at least one of these tasks. Thus, for programs with rare concurrency bugs, most tasks do not contain a bug. However, in tasks that do, the bugs are generally more frequent (i.e., manifest in a higher fraction of the interleavings) than in the original program. Consequently, bugs can be found faster and with fewer resources, because the individual tasks are simpler and can be analyzed in parallel,

each with a shorter time-out and a smaller memory.

We develop and evaluate this approach under a bounded context-switch analysis where only interleavings with up to k context-switches (for a given k) are explored. This choice is justified by an empirical study which shows that most of the concurrency bugs manifest themselves within a small number of context-switches [9]. We use a code-to-code translation to derive the tasks as variants of the original program by splitting the code of each thread into fragments (*tiles*) and allowing context-switches only in some of them. By selecting $\lceil \frac{k}{2} \rceil$ tiles in all possible ways, we thus ensure the coverage of all interleavings up to k context-switches.

Our approach offers a number of advantages. First, since it is using code-to-code translations, it is "agnostic" of the underlying verification techniques and tools: existing bug-finding tools can be reused as is, and while we have achieved very good results using bounded model checking (BMC) techniques (in particular Lazy-CSeq [10], [11]), it can also be used with other symbolic analysis techniques, explicit state space exploration techniques, or even testing.

Second, our approach amplifies the effectiveness of existing bug-finding tools. We empirically demonstrate that it is particularly effective for symbolic methods: it reduces the memory consumption and run time of each individual verification task containing a bug, and also leads to a considerable reduction in time for the global verification. We show a substantial reduction in the wall-clock times required to find a bug in some very difficult problems: from 8-12 hours using a single instance of Lazy-CSeq, the only tool capable of finding the underlying bugs, down to 15-30 minutes using Lazy-CSeq on a modest number (5-50) of processors. Looking at this from an opposite perspective, our approach enables existing tools to find rare bugs that were previously out of their reach.

Third, our approach also is oblivious of the assumed memory model and therefore also works for WMMs, as long as the underlying analysis tool supports their semantics. Moreover, our experiments demonstrate that the approach is also effective for reducing the additional verification complexity introduced by relaxed consistency semantics.

Finally, our approach is tuneable. The verification complexity of each of the instances generally depends on the underlying analysis tool and number of interleavings captured by each instance. From our experience, instances with roughly the same number of interleavings have similar verification times. Thus, we empirically learn the number of interleavings per instance that the underlying tool can effortlessly handle and then generate all the instances to capture all interleavings according to a fixed schema. However, the number of instances to generate can be extremely high. We empirically show that using only a few instances selected at random still enables us to find rare bugs. In our experiments, we demonstrate that we only have to consider a few instances (out of millions or billions) to find bugs with high probability.

Contributions. In summary, in this paper we make three main contributions. First, we propose a new swarm verification approach for the analysis of concurrent programs that is based on a code-to-code translation and leverages the power of sequential analysis engines. Second, we implement the approach as an extension to Lazy-CSeq. Third, we report the results of an evaluation of our approach on the two hardest known concurrency benchmarks, `safestack` [12] and `eliminationstack` [13] for three different memory models (SC, TSO, PSO).

Organization of the paper. In the next section, we give a high-level overview of our approach. Sections III and IV describe our code-to-code translation. Section V gives details on our implementation and Section VI presents the results of our experimental evaluation. Section VII compares with related work, and Section VIII concludes.

II. Approach

We consider multi-threaded programs where threads communicate through shared memory (for example, a C program that uses the POSIX threads library for concurrency). As in bounded model checking, we first flatten the program by inlining functions and unrolling loops up to a given bound. The resulting *bounded* program, say P, consists of a finite number of threads; the control in each thread can only move down in the code. The goal of the analysis is to find an assertion violation of P that may occur through an execution that involves at most k context-switches (for a given k). Let k be a small natural number denoting the maximum number of context-switches to consider along an execution, and T be the set of P's threads. We denote with $\mathcal{I}_k(P)$ the set of all executions that P can exhibit with at most k context-switches.

A. Splitting Computations with Tilings

Our goal is to define a code-to-code translation for P, parameterized over k, that generates a set of simpler program variants, each capturing a subset of P's executions, such that each of P's executions involving at most k context-switches is captured by at least one of them. The resulting variants can then be checked independently in parallel.

We construct these variants by building on the notion of tiling. A *tiling of a thread* $t \in T$ is a partition of t's statements. Each element of a tiling is called *tile*. For example, consider Fig. 1. The program has two threads with respectively seven (*A, ..., G*) and five (*H, ..., L*) statements, and the tiling of each thread is marked with the braces. A *tiling of a program* P is a set of thread tilings, one for each thread of P. Let $\Theta_P = \{\Theta_t\}_{t \in T}$ be a tiling of P. A *z-selection* of Θ_P is a set $\{\theta_t\}_{t \in T}$ where $\theta_t \subseteq \Theta_t$ contains exactly z tiles.

We build the variants as follows. For a given tiling Θ_P of P and any of its z-selections $\vartheta = \{\theta_t\}_{t \in T}$, we construct a program variant P_ϑ of P obtained by instrumenting P in a way that each thread t can only be preempted at statements belonging to the tiles of θ_t and at any other blocking statement of t (in order to allow an execution to continue when a statement is blocked and there are other threads that are ready to execute). Consider again Fig. 1. If we take the 1-selection ϑ corresponding to selecting tiles #1 and #4, the executions of P_ϑ are of the form uvw where: (1) u is any interleaving of

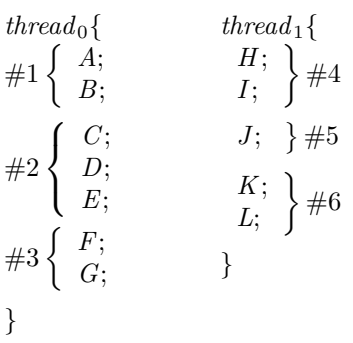

Fig. 1. Tiling example

A-B and *H-I*; (2) if u ends with *B*, then v = *C-...-G* and w = *J-K-L*; (3) if u ends with *I*, then v = *J-K-L* and w = *C-...-G*. For example, *A-B-H-...-L-C-...-G* and *A-H-I-B-C-...-G-J-K-L* denote possible executions of P_ϑ.

We observe that the set of executions over all the variants P_ϑ, with ϑ being a $\lceil \frac{k}{2} \rceil$-selection of Θ_P, that contain at most k context-switches is exactly the set $\mathcal{I}_k(P)$. In fact, every execution of a variant P_ϑ of P is also an execution of P since P_ϑ, by construction, is the same as P, except that we forbid context-switches to occur at some points. Vice-versa, any execution $\pi \in \mathcal{I}_k(P)$ can context-switch out of each thread at most $\lceil \frac{k}{2} \rceil$ times, thus it suffices to select $\lceil \frac{k}{2} \rceil$ tiles per thread to capture π. Therefore, π is an execution of some variant P_ϑ, for a $\lceil \frac{k}{2} \rceil$-selection ϑ.

B. Overall Approach

The verification approach works in two phases. We first generate P's variants according to any selection of an input tiling. We then search for bugs in each of the resulting program variants (typically in parallel) using an off-the-shelf analyzer such as a testing tool or a model checker. The analysis phase can be stopped as soon as a bug is found in one of the program variants. The overall scheme is shown in Fig. 2; in the following, we sketch the two phases in turn.

1) Instance Generation: The first phase is composed of a chain of code-to-code transformations of the input multi-threaded program.

The first module transforms this program into a bounded multi-threaded program P that is syntactically guaranteed to terminate after a bounded number of transitions, by applying standard BMC program transformations [14] such as inlining the functions and unwinding the loops (up to a given bound). P thus has a different function associated with each thread; we refer to these as *thread functions*.

The second module injects numerical labels at each *visible* statement of the thread functions (i.e., at the beginning and end of the function, before each access to the shared memory, and before each call to a thread synchronization primitive) of P. In each thread function, the labels start at zero and increase consecutively in statement order; we assume that any other label of the program is non-numerical. This labeling simplifies the code injected by the third module for the tile selection.

The third module instruments the code with guarded commands that at each numerical label enable/disable context-switchs and statement reordering. This control-flow code is used by the next module to imprint the tile selection in the code. A detailed description of the translation by this module is given in Section IV.

The fourth and last module generates the variants of P according to any z-selection of the input tiling Θ_P (where z is the value of the input parameter #tiles). This is done by triggering the guards injected by the previous module. Note that the number of different program variants generated this way is finite, but can be large. Therefore, we consider a randomized version of this module along with a new input parameter, the number n of instances to be generated. The n instances are generated by *randomly* choosing the z-selections. This, also introduces a *loop* in our verification approach: we repeat the random generation of n new variants until either a bug is found or all the variants are generated.

2) Verification Cluster: Since the generated instances can be analyzed independently, we can achieve in our scheme a high diversification and parallelism of the analysis: each instance can be analyzed on a separate core and possibly using a different tool for concurrent program analysis.

III. Shared-memory Multi-threaded Programs

Our implementation can handle the full C language (see Section V), but we describe our approach for multi-threaded programs in a simple imperative language. This features dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. Thread communication is implemented via shared memory and modelled by global variables. All threads share the same address space: they can write to or read from global variables of the program to communicate with each other. We assume that each statement is atomic as it is always possible to decompose a statement into a sequence of statements each involving at most one access to the shared memory [15].

A. Syntax

The syntax of multi-threaded programs is defined by the grammar shown in Fig. 3. Terminal symbols are set in typewriter font. Notation $\langle n\ \mathtt{t} \rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals $\mathtt{t}$; x denotes a local variable, y a shared variable, t a thread variable and p a procedure name. All variables involved in a sequential statement are local. We assume expressions e to be local variables, integer constants, that can be combined using mathematical operators. Boolean expressions b can be `true` or `false`, or Boolean variables, which can be combined using standard Boolean operations.

A *multi-threaded* program consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of procedures. Each procedure has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement. A statement is either a simple statement or a compound statement, i.e., a sequence

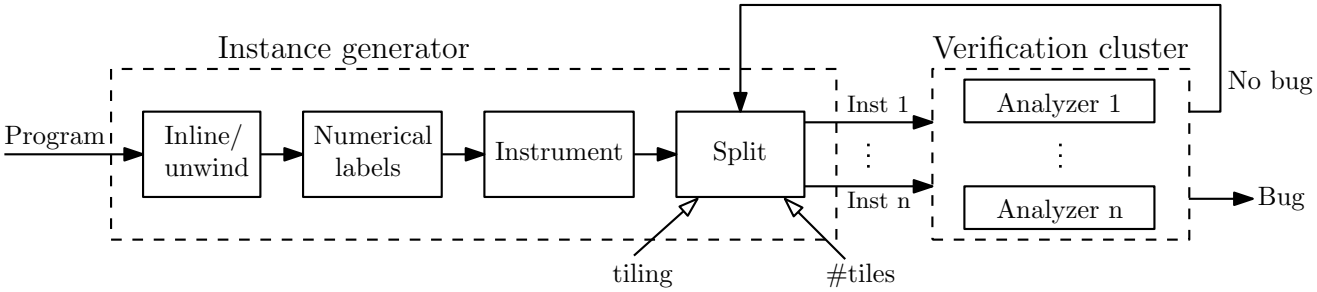

Fig. 2. Verification approach.

$$
\begin{array}{lcl}
P & ::= & (dec\texttt{;})^*\ (type\ p\ \texttt{(}\langle dec\texttt{,}\rangle^*\texttt{)}\ \ \texttt{\{}(dec\texttt{;})^* stm\texttt{\}})^* \\
dec & ::= & type\ z \\
type & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
stm & ::= & sstm \mid \texttt{\{}\langle stm\texttt{;}\rangle^*\texttt{\}} \\
sstm & ::= & seq \mid conc \mid l\texttt{:}\ sstm \\
seq & ::= & \texttt{assume}(b) \mid \texttt{assert}(b) \mid x := e \mid p\,\texttt{(}\langle e\texttt{,}\rangle^*\texttt{)} \\
 & & \mid \texttt{return}\ e \mid\ \texttt{if}(b)\,\texttt{then}\ stm\ \texttt{else}\ stm \\
 & & \mid \texttt{while}(b)\ \texttt{do}\ stm \mid \texttt{goto}\ l \\
conc & ::= & x := y \mid y := x \mid t := \texttt{create}\ p(\langle e\texttt{,}\rangle^*) \\
 & & \mid \texttt{join}\ t \mid\ \texttt{init}\ m \mid\ \texttt{lock}\ m \\
 & & \mid \texttt{unlock}\ m \mid \texttt{destroy}\ m
\end{array}
$$

Fig. 3. Syntax of multi-threaded programs.

of statements enclosed in braces. A simple statement is either a labeled simple statement, or a sequential statement, or a concurrent statement.

A *sequential statement* can be an `assume`- or `assert`-statement, an assignment, a call to a procedure that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, a `while`-loop, or a jump to a label. Local variables are considered uninitialized right after their declaration, which means that they can nondeterministically take any value from their type domains until they are explicitly set by an assignment statement. We also use the symbol $\star$ to denote the expression that nondeterministically evaluates to any possible value; with $x\texttt{:=}\star$ we then mean that x is assigned with any possible value of its type domain.

A *concurrent statement* can be a concurrent assignment, a call to a thread routine, such as a thread creation, a join, or a mutex operation (i.e., init, lock, unlock, and destroy). A concurrent assignment assigns a shared (resp. local) variable to a local (resp. shared) one. Unlike local variables, global variables are always assumed to be initialized to a default value. For the sake of simplicity, we assume that the default value always `0` regardless of the variable type. A thread creation statement $t := \texttt{create}\ p(e_1, \ldots, e_n)$ spawns a new thread from procedure p with expressions $e_1, \ldots, e_n$ as arguments. A thread join statement, $\texttt{join}\ t$, pauses the current thread until the thread identified by t terminates its execution, i.e., after the thread has executed its last statement. Lock and unlock statements respectively acquire and release a mutex. If the mutex is already acquired, the lock operation is blocking for the thread, i.e., the thread is suspended until the mutex is released and can then be acquired.

We assume that a valid program P satisfies the usual well-formedness and type-correctness conditions. We also assume that P contains a procedure `main`, which is the starting procedure of the only thread that exists in the beginning. We call this the *main thread*. We further assume that there are no calls to `main` in P and that no other thread can be created that uses `main` as starting procedure.

B. Semantics

As common, a program configuration is a tuple of configurations of each thread that has been created and has not yet terminated, along with a valuation of the global variables. A thread configuration consists of a *stack* which stores the history of positions at which calls were made, along with valuations for local variables, and the top of the stack contains the local and global valuations, and a pointer to the current statement being executed.

The behavioral semantics of a program P is obtained by interleaving the behaviors of its threads. At the beginning of any computation only the main thread is *ready* and *running*. At any point of a computation, only one of the ready threads is *running*. A step is either the execution of a step of the running thread or a *context-switch* that nondeterministically replaces the running thread with one of the ready ones that thus becomes the running thread at the next step. A thread will no longer be available when its execution is terminated, i.e., there are no more steps that it can take.

IV. Code-to-code translation

In this section, we give a formal description of the code instrumentation done by module *Instrument* in Fig. 2.

For the sake of better presentation, in the following, in order to enable/disable context-switches in the code, we consider a semantics in the style of *preemptive asynchronous programs*

$$
\left[\!\!\left[\begin{array}{l} (dec;)^*\ (type\ p_i\ (\langle dec, \rangle^*) \\ \{(dec;)^* stm\})_{i=0,\ldots,n-1} \end{array} \right]\!\!\right] \stackrel{\text{def}}{=} \begin{array}{l} \texttt{bool yields[n][h]} = \{\{a_0^0, a_1^0, ..., a_{h-1}^0\}, \ldots, \{a_0^{n-1}, a_1^{n-1}, ..., a_{h-1}^{n-1}\}\}; \\ (dec;)^*\ (type\ p_i\ (\langle dec, \rangle^*)\ \{(dec;)^* [\![stm]\!]_i\})_{i=0,\ldots,n-1} \end{array}
$$

$$
[\![stm]\!]_i \stackrel{\text{def}}{=} [\![sstm]\!]_i \mid \{\langle [\![stm]\!]_i ; \rangle^*\}
$$

$$
[\![sstm]\!]_i \stackrel{\text{def}}{=} seq \mid conc \mid [\![l\colon sstm]\!]_i
$$

$$
[\![l\colon sstm]\!]_i \stackrel{\text{def}}{=} \begin{cases} l\texttt{: if(yields[}i\texttt{][}l\texttt{] \&\& *) yield;} \\ \quad [\![sstm]\!]_i\texttt{;} & \text{if } l \text{ is numerical} \\ l\colon [\![sstm]\!]_i & \text{otherwise} \end{cases}
$$

Fig. 4. Formal description of the code-to-code translation by module *Instrument*.

with nondeterministic scheduler. In particular, we augment the concurrent statements of the syntax from Fig. 3 with a `yield`-statement, i.e., we add the rule $conc ::= \texttt{yield}$, and restrict the context-switches to occur only if explicitly invoked via a `yield`-statement (which thus causes the control to return to the nondeterministic scheduler). In the following, we will refer to this class of programs as *extended preemptive asyncronous programs* (EPA programs, for short).

It is straightforward to show that given a multi-threaded program P under the syntax and semantics of Section III we can easily obtain an equivalent program P' under syntax and semantics sketched above by simply inserting a `yield`-statement guarded by a nondeterministic guess in front of each statement of P. Moreover, as we are interested only in reachability of program locations or assertion failure checking, such as in standard bug-finding analysis, it is sufficient to account only for context-switches that occur at visible statements (i.e., the concurrent statements and each thread's first and last statements).

We recall that we assume that all the labels of the original multi-threaded program must be non-numerical and that after the code-to-code translation by modules *Inline/unwind* and *Numerical labels*, we get that: (1) the code of each thread is all contained within the same procedure, i.e., there are no procedure calls, and there are no loops; and (2) the visible statements are all labeled with a numerical label such that in each thread code labels start from `0` and increase by `1` according to the statement order.

In module *Instrument*, we thus rewrite the code by inserting a guarded `yield`-statement after each numerical label. Guards are triggered by input Boolean parameters: we use a_l^i to activate the `yield`-statement at the numerical label l of thread i. These parameters are assigned to a Boolean array `yields`.

After the instrumentation, the portion of code $l\texttt{:}\, sstm$ of thread i, where l is a numerical label, is thus:

`if(yields[`i`][`l`] && *) yield;` $sstm$`;`

The rest of the code stays unchanged.

We formally give our code-to-code translation in Fig. 4 as rewrite rules over the syntax of programs. In the figure, we have denoted with `n` the number of threads and with `h` the maximum number of numerical labels over all threads.

We observe that whenever `yields[`i`][`l`]` holds, by the choice operator $\star$, the `yield`-statement is nondeterministically executed or not. This can be used to select in the code the points where context-switches can happen. Thus, the following module *Split* (see Fig. 2) will assign the array `yields` accordingly to a valid selection for the input tiling and blocking statements.

V. Implementation

We have implemented the verification approach illustrated in Fig. 2 to analyze concurrent C programs that use the concurrency library POSIX Threads. We optimize the tilings by taking into account only statements at which context-switches can occur which correspond to numerical labels. For the instance generation we use *uniform window tilings*, i.e., tilings where all tiles have the same number of numerical labels except for the last one that can have fewer, and all tiles correspond to contiguous sections of a thread's code. For example, the tiling from Fig. 1 is not uniform though tiles cover contiguous portions of code.

The *Instance generator* is written as an independent component that takes as input: (1) a multi-threaded program P with assertions, (2) the unwinding bound, (3) the size t of each tile (i.e., the number of numerical labels), (4) the number s of tiles to select from each thread, and (5) the number n of randomly chosen instances to generate. The pool of (bounded) EPA programs generated by the *Instance generator* is then verified on a cluster of computers with a modified version of the symbolic verification tool Lazy-CSeq [10].

Below we provide more details on our implementation of the *Instance generator*, the verification tool, and the cluster.

A. Instance Generator

Our tool VeriSmart[1] ("Verification-Smart"), builds upon the CSeq framework [10], [16], and is composed of a chain

[1] VeriSmart is publicly available at http://users.ecs.soton.ac.uk/gp4/cseq/.

of software modules that matches the chain of modules of the *Instance generator* from Fig. 2. We recall that CSeq is a framework that comprises several software modules implementing standard source-to-source transformations of C programs. We re-use CSeq modules to implement the modules *Inline/unwind* and *Numerical labels*. For *Instrument*, we have realized a new software module that implements the code-to-code translation detailed in Section IV. It is written in Python and uses the AST built by pycparser[2] on the fetched program to implement the rewriting rules of Fig. 4. The last module *Split* is also written in Python. It generates each instance by randomly selecting s tiles per thread by setting to true the corresponding entries of the array `yields` (entries corresponding to blocking statements are always set to false). This module also takes an additional parameter that allows to bound the number of generated instances. In our setting we manually allocate the resulting instances to several verification units that are analyzed independently.

The code of VERISMART is publicly available at http://users.ecs.soton.ac.uk/gp4/cseq/ and can be used, in combination with different bug-finding analysis tools, for experimenting with the verification approach proposed in this paper.

B. Backend Verification Tool

Lazy-CSeq [10] is a symbolic bug-finding tool for multi-threaded programs based on sequentialization and bounded model checking. The multi-threaded input program is translated into a corresponding sequential program up to a given number of rounds of execution, where in each round each thread is executed exactly once according to a fixed ordering. The resulting sequential program is then verified using existing verification tools for sequential programs.

We modify Lazy-CSeq to account for the syntax and the semantics of EPA programs. Essentially, we extend the parsing for handling also the `yield`-statement and then adjust the code-to-code translation such that in the resulting sequential C program the context-switches are now simulated only at the `yield`-statements according to the semantics of the EPA programs. In the following, we refer to this modified version of Lazy-CSeq as EPA-Lazy-CSeq.

In our experiments we use the C bounded model checker CBMC[3] v5.3 [17], [18] as sequential backend for Lazy-CSeq and EPA-Lazy-CSeq. CBMC encodes symbolically the multiple execution paths of the input program, which is then checked by a SAT/SMT solver. If the formula is satisfiable there is a definite execution path that leads to an assertion violation. In this case the SAT/SMT solver returns the values of the variables along this execution path. From these values, which in particular include the input values and the encoded context switch points, a test can be constructed and executed in order to debug the reason for the assertion violation. If the formula is unsatisfiable then there is no execution that violates any of the assertions (up to the considered depth). The formula is generated by symbolically executing the program while encoding the control flow structure into additional Boolean variables. The formula is linear in the size of the program, but implicitly encodes a potentially exponential number of execution paths. Hence, unlike path-wise enumeration approaches [19], [20] this approach avoids explicitly enumerating a potentially exponential number of execution paths and maximizes the exploitation of today's optimized SAT solvers.

Lazy-CSeq was initially developed for C program running under the SC semantics. It has been recently extended to handle WMM semantics such as TSO and PSO [21]. Our modification are also compatible with these extensions. In our experiments we only use EPA-Lazy-CSeq for the analysis carried out on the cluster. Our choice has been motivated by the effectiveness of Lazy-CSeq for complex benchmarks containing rare bugs.

VI. EXPERIMENTS

Here we report on a large number of experiments that we have conducted to demonstrate the effectiveness of the proposed approach. It turns out that VERISMART improves the performance of Lazy-CSeq by achieving more than 1000x speed-up already with few cores (≤ 20) on some benchmarks that are hard to analyze.

A. Benchmarks

Our first effort was to identify suitable benchmarks that present a non-trivial challenge for concurrency bug-finding tools. The widely used concurrency benchmarks from the Software Verification Competition (SV-COMP) are too easy: several tools can quickly find the bugs in all buggy benchmarks; Lazy-CSeq requires 2.5 seconds on average.[4]

We have instead considered several concurrency benchmarks from the literature, including the SCT Benchmarks[5] [22]. From these, we discarded all parametric benchmarks whose complexity comes simply from artificially increasing the number of threads (e.g., `CS.reorder_`n`_bad` and `CS.twostage_`n`_bad` [22], where n is the number of threads), or the size of the data structures (e.g., the work stealing queue [22]), since their bugs can already be exposed with smaller instances. We further discarded all benchmarks that Lazy-CSeq can solve in less than 600 seconds; this includes several benchmarks that are traditionally considered to be hard, e.g., DCAS [23]. Similarly to the SV-COMP benchmarks, our experiments showed that there is no significant performance difference between Lazy-CSeq and VERISMART—in some sense these benchmarks are "too simple" to benefit substantially from the VERISMART approach. We therefore do not

[2]pycparser: https://github.com/eliben/pycparser

[3]CBMC: http://www.cprover.org/cbmc/

[4]For the hardest benchmark (fibonacci), Lazy-CSeq required less than 300 seconds, but this is an atypical, artificial example specifically crafted to require a high number of context switches (21) in a small program (11 visible statements in two threads each), which is smaller than our usual tile size, so that there is no difference to VERISMART. For four other benchmarks, Lazy-CSeq requries between 20 and 90 seconds; here, VERISMART performs similarly. For the majority of the simple benchmarks, VERISMART pays a small wall-clock time penalty due to the transformation overhead.

[5]SCT Benchmarks: https://sites.google.com/site/sctbenchmarks/

report further details for these benchmarks. This leaves us with two benchmarks that both come from the domain of concurrent data structures.[6]

`eliminationstack` is a C implementation of Hendler et al.'s Elimination Stack [13] that follows the original pseudocode presentation. It augments Treiber's stack with a "collision array", used when an optimistic push or pop detects a conflicting operation; the collision array pairs together concurrent push and pop operations to "eliminate" them without affecting the underlying data structure. This implementation is incorrect if memory is freed in pop operations. In particular, if memory is freed only during the "elimination" phase, then exhibiting a violation (an instance of the infamous ABA problem) requires a seven thread client with three push operations concurrent with four pops. To witness the violation, the implementation is annotated with several assertions that manipulate counters as described in [24]. Lazy-CSeq is the only tool we are aware of that can automatically find bugs in this benchmark and requires almost 13 hours and 4 GB of memory to find a bug.

`safestack` is a real world benchmark implementing a lock-free stack designed for weak-memory models. It was posted to the CHESS forum[7] by Dmitry Vyukov. This benchmark is unique as it contains a very rare bug that requires at least three threads and five context-switches to get exposed when running under the SC semantics, whereas it requires only four context-switches when running under TSO or PSO. In the verification literature, it was shown that real-world bugs require at most three context-switches to manifest themselves [9]. `safestack`, for this reason, presents a nontrivial challenge for concurrency testing and symbolic tools. Lazy-CSeq is the only tool we are aware of that can automatically find these bugs: it takes almost 7 hours to find these bugs and consumes more than 6 GB of memory.[8]

B. Experimental Set-Up

In our experimental evaluation we compare VERISMART against Lazy-CSeq v1.0.[9] The results are summarized in Fig. 5.

For both tools, we use the minimum number of unwindings and rounds of computation that are required to expose the bug in the original program by Lazy-CSeq; the exact values for these parameters are reported in Fig. 5 for each considered benchmark. We also report the number of the threads involved in the computation and the number of visible points in each of those threads.

For Lazy-CSeq, the results for the comprehensive verification are given in the leftmost columns of Fig. 5. Under "All schedules", we report the verification time and memory consumption when all interleavings are analyzed in one attempt. We also estimate the "sequential complexity" of the benchmarks as the average performance to analyze them given one fixed schedule. This gives us an estimated lower bound for the analysis of each of the tasks generated by VERISMART. For this, we have considered 3000 randomly selected interleavings. We report the computed average time and memory consumption in Fig. 5 under "One schedule".

For VERISMART, see Fig. 5 again, we select as many tiles per thread as the number of rounds required to expose the bug in each of the considered benchmark; this ensures that the bug can (in principle) be found. We use a uniform window tiling with three different tile sizes to evaluate the effect of tiling. For each benchmark and tile size, we then analyze 8,000 program variants with the timeout given in Fig. 5; these variants are generated by randomly selecting the tiles. We report the minimum, the maximum, the average and the standard deviation over the verification time and memory consumption of all the buggy variants. We also report the percentage of the buggy variants.

We also estimate, for each considered benchmark and setting (using the data collected in the VERISMART experiments described above), how the expected bug-finding time varies as the number n of cores increases. We thus randomly draw sets of n variants containing at least one with a bug, and compute the the minimum time to find a bug over this set. We then plot the expected bug-finding time for values of n up to 2000 (see Fig. 6). Here, a round (resp. square) point marks the expected number of cores required find a bug with probability 0.95 (resp. 0.99) and the corresponding expected time.

We carried out our experiments on a cluster with 750 compute nodes equipped with dual 2.6 GHz Intel Sandybridge processors. Each compute node has 16 CPUs with 64 GB of physical shared memory running 64-bit linux 3.0.6. On each CPU of a node we run EPA-Lazy-CSeq over a single verification task produced by the instance generator, with the timeout given in Fig. 5.

C. Experimental Results

Fig. 5 and 6 give detailed results of our experiments; the `safestack`-TSO results are summarized in the text.

As expected, the results for the comprehensive verification using Lazy-CSeq show that both benchmarks are very hard under SC. The relation between the numbers for one schedule and all schedules, respectively, shows that the complexity comes from the interleavings. Less expectedly, with Lazy-CSeq requiring 11005 seconds and 4.3 GB of memory to find a bug in `safestack`-TSO, they also show that WMMs can actually make it easier to find bugs, as both runtime and memory requirements go down. This is a consequence of both the lower number of rounds required to expose the bug, and the higher number of buggy executions that the WMMs allow.

The main part of Fig. 5 shows the VERISMART results for the different tile sizes. We focus on the SC benchmarks first, and defer the discussion of WMM benchmarks to below.

[6]It is hardly surprising that lock-free programming is an important source of truly *concurrently complex* benchmarks since the focus there is to *minimize* the amount of synchronization for performance optimization thus generating a large amount of nondeterminism due to interleaving.

[7]https://social.msdn.microsoft.com/Forums/en-US/91c1971c-519f-4ad2-816d-149e6b2fd916/bug-with-a-context-switch-bound-5?forum=chess

[8]The tool Relacy [22] can find a bug in a modified version of `safestack` where an explicit `pthread_yield` call has been added to help the search. In our experiments, on the plain `safestack` benchmark used here, Relacy was not able to detect a bug within one million iterations.

[9]CSeq framework: http://users.ecs.soton.ac.uk/gp4/cseq

`eliminationstack`-SC (unwind=1, rounds=2, thread=8, visible point=52)

COMPREHENSIVE VERIFICATION			VERISMART: 2 tiles per thread								
	Time	Memory	#1: tile size 12, timeout 1.5hrs			#2: tile size 14, timeout 2hrs			#3: tile size 18, timeout 3hrs		
			Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
One schedule	80.8	661.1	Min	34.9	945.2	Min	39.7	979.84	Min	37.1	999.8
			Max	4753.6	1199.1	Max	7195.2	1281.3	Max	10762.0	1785.5
			Average	1116.3	1017.8	Average	2169.5	1096.3	Average	3162.4	1156.9
All schedules	46764	4203.9	Deviation	1051.4	32.4	Deviation	1935.3	68.9	Deviation	2699.6	133.0
			instances with bug: 38.33%			instances with bug: 61.38%			instances with bug: 69.01%		

`safestack`-SC (unwind=3, rounds=4, thread=4, visible point=152)

COMPREHENSIVE VERIFICATION			VERISMART: 4 tiles per thread								
	Time	Memory	#1: tile size 11, timeout 1hr			#2: tile size 14, timeout 1hr			#3: tile size 20, timeout 4hrs		
			Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
One schedule	55.4	700.1	Min	195.6	774.5	Min	574.8	846.6	Min	313.0	850.3
			Max	2662.6	1265.7	Max	3521.8	1450.4	Max	10315.8	3830.8
			Average	1172.2	928.8	Average	1851.1	1147.3	Average	2167.5	1230.1
All schedules	24139	6632.4	Deviation	552.4	122.5	Deviation	827.3	165.5	Deviation	1558.1	317.1
			instances with bug: 1.26%			instances with bug: 2.14%			instances with bug: 10.20%		

`safestack`-PSO (unwind=3, rounds=3, thread=4, visible point=152)

COMPREHENSIVE VERIFICATION			VERISMART: 3 tiles per thread								
	Time	Memory	#1: tile size 12, timeout 1.5hrs			#2: tile size 16, timeout 2.25hrs			#3: tile size 20, timeout 3hrs		
			Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
One schedule	272.5	2651.8	Min	898.1	2795.9	Min	593.5	2862.6	Min	1083.8	2910.1
			Max	5348.0	3280.7	Max	8083.1	3942.2	Max	10771.7	3784.1
			Average	2929.8	2872.9	Average	4607.1	3015.4	Average	5176.9	3073.8
All schedules	4777	3708.4	Deviation	941.4	52.4	Deviation	1751.7	154.2	Deviation	2265.9	145.9
			instances with bug: 7.63%			instances with bug: 16.73%			instances with bug: 26.85%		

Fig. 5. VERISMART experiments: each experiment is carried out using 8,000 instances chosen randomly. The verification is done using Lazy-CSeq 1.0 for all schedules, and EPA-Lazy-CSeq for the program variants. For each experiment, we report the minimum, the maximum, the average and the standard deviation over the verification time and memory consumption of all the buggy variants. Time is given in seconds and memory in MB.

For SC, we see the best-case resource consumption required to expose a bug in any of the generated instances drops dramatically: the best instances expose the bugs 100x to 1000x faster, with only 10% to 25% of the memory. The relative numbers of buggy instances are very favorable and can be improved even further, by increasing the tile sizes. These increased odds come at modest costs: average times to expose the bugs increase 2x–3x, while average memory consumption remains roughly stable. Taken together, this means that we only need to analyze a small number (less than 100, and in many cases less than 10) of relatively simple (average times to find the bugs between 20 minutes and one hour) problems to find a bug with probability approximating 1 (see Fig. 6).

For `safestack`-PSO the results appear at first less impressive. While the best-case bug-finding times still represent a roughly 5x speed-up, the average times are closer to (and in some times even exceed) the comprehensive analysis times. However, the high fraction of buggy instances (roughly 5%–25%) allows us to play the numbers game to achieve a good overall performance. If we run a moderate number (say 50) of tasks in parallel, we will with high probability (since the distribution of the bug-finding times exhibits a log-normal shape) also come across one of the "faster" tasks, which will abort the remaining "slower" tasks, giving us wall-clock speed-ups roughly similar to the best cases.

Finally, for `safestack`-TSO none of the 24,000 tasks generated with three selected tiles exposes a bug within the given timeouts (despite the fact that this already represents 52,000 hours CPU-time). Since the experimental set-up means we are only looking for bugs that occur only under TSO but not under SC, we can clearly see that the TSO-only bug is extremely rare. In such cases, VERISMART is unable to leverage the effectiveness of the underlying analysis tool, and suffers from the same explosion of the search spaces as other sampling-based methods such as testing or explicit state model checking. Here, symbolic methods that analyze all behaviors simultaneously have the upper hand, as demonstrated by Lazy-CSeq's ability to expose the TSO-only bug.

Since the generated problem instances are completely independent and can be analyzed in parallel, VERISMART is indeed a very effective verification approach. Fig. 6 shows that with moderate resources (5–50 cores) we can get a noticeable speed-up on the expected bug-finding time. In particular, for `eliminationstack`-SC the expected time is roughly the same as the minimum time already for few cores and reaches a more than 1000x speed-up compared to the all-schedule comprehensive verification time.

In general, Fig. 6 shows that the expected bug-finding time converges faster to the minimum time when the probability of finding a buggy program variant is higher. Since larger tiles would ensure higher probability of finding a buggy variant but at the same time could affect the verification time, determining the right tile size seems to be a crucial aspect for maximizing the benefits of our approach.

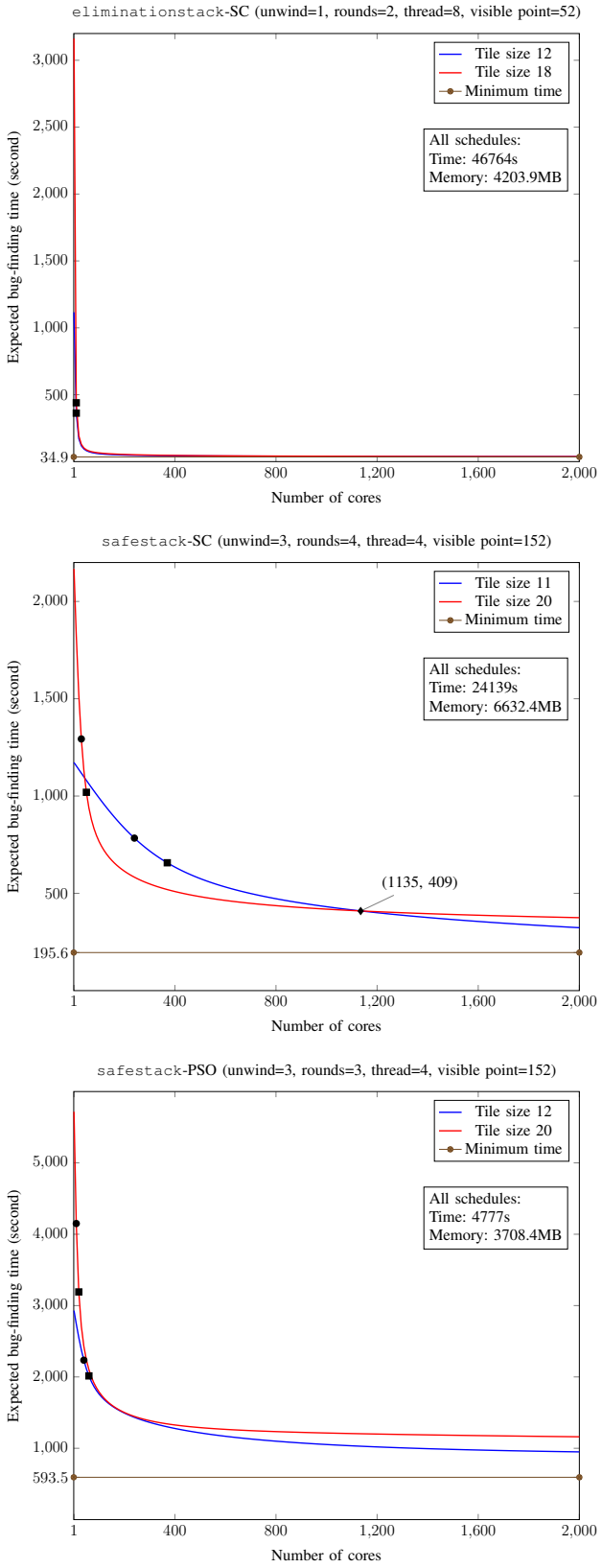

Fig. 6. Expected bug-finding time varying over the number of cores. A round (resp. square) point marks the expected time corresponding to the number of cores that are needed to find a bug with probability 0.95 (resp. 0.99).

VII. Related work

There is a wide range of approaches proposed in the literature on automatic analysis of concurrent programs. Here we briefly describe the related work and compare it to the work presented in this paper.

Parallel Verification: Attempts to parallelize verification by partitioning the problem and distributing the workload have been implemented in explicit-state model checking [2], [25] and SAT solving [3]. With the rise of multi-core processors, techniques that exploit shared memory for communication have been proposed [26], [27]. However, these approaches suffer from the overhead introduced by exchanging information between the instances. Approaches that run several tools with different strategies and heuristics in parallel on the unpartitioned problem have been more successful. Such *portfolio* approaches have been implemented in automated theorem provers [6], [7], [8] and SAT/SMT solvers [5], [28]. Our approach leverages so-called *swarm verification* (SV), as promoted by Holzmann et al for explicit-state model checking [4], [1], [29]. In SV computing instances do not collaborate directly in finding a solution, but solve independent subproblems that cover the original problem. We lift this idea to symbolic model checking through sequentialization.

Sequentialization: Reducing the analysis of a concurrent program to the analysis of sequential programs was first proposed by Qadeer and Wu [30]. They transform a concurrent program into a sequential one that simulates all executions of the original program with at most two context-switches. Lal and Reps [31], [32] generalized the concept to arbitrary context bounds. In our experiments, we used Lazy-CSeq [10], [33], [11], [34], [35], which implements a sequentialization as a code-to-code translation that is efficiently analyzable by sequential bounded model checking tools such as CBMC [18]. Musuvathi and Qadeer [9] propose an algorithm for iteratively relaxing the context bound. This is orthogonal to our approach: in our experiments we fixed the maximum number of context-switches and analysed tilings of three different sizes. We could consider their algorithm as a starting point to automatically find good parameter values for our tilings.

Concolic Testing: Concolic testing [19] combines symbolic aspects with concrete inputs. Namely, it runs the program over an input vector with both concrete and symbolic values, and uses SMT solvers to compute new input vectors that systematically explore the branches of the program. Farzan et al [36] extend this idea to concurrent programs and call it (con)2colic testing. They use the notion of thread interference scenario, which is a representation of a set of bounded interferences [37] among the threads, which define the scheduling constraints for a concurrent program run. These interference scenarios are then explored in a systematic way by generating a schedule and input vectors that conform with the scenario. (con)2colic testing analyzes the program sequentially and accumulates information about explored scenarios in a data structure, whereas our approach is capable of analyzing tilings *independently* and can thus be parallelized at large scale.

Moreover, (con)2colic testing requires to modify the core of a concolic testing tool in order to handle concurrent programs and thus it cannot flexibly leverage the increasing power of existing sequential checkers.

Testing: Automated testing tools such as CHESS [38] have been highly successful for finding concurrency bugs in large code bases because of their ability to handle code independently of its sequential complexity. CHESS controls the scheduler and explores all possible interleavings giving priority to schedules with few context-switches. Nonetheless, the success of testing depends on the proportion of schedules that lead to a bug w.r.t. the total number of schedules, as shown by a recent empirical study [39], [22] on testing of concurrent programs. Preemption sealing [40] consists of inhibiting preemptions in some program modules which corresponds in our approach to choose a tiling where tiles exactly correspond to program modules. This strategy was aimed to tolerating errors for finding more ones and compositional testing of layered concurrent systems. The uniform tiling we implement in this paper is irrespective of the structure of the program and looks more appropriate for an exhaustive bug-finding search up to a given number of context-switches. There are also differences in the implementation of the two techniques, we do not seal portions of code with scope functions but rather we implement tiles statically, that in general makes the underlying BMC analysis simpler. Other testing tools try to mutate observed interleavings to find bugs (e.g. [41], [42]). Our approach can be seen as a way to tune concurrency verification between concrete testing and fully symbolic verification.

VIII. Conclusions and Future Directions

We have presented a swarm approach for finding bugs in concurrent programs. We perform a code-to-code translation that constructs program variants by placing tiles over the threads, and thus reducing nondeterminism by allowing context-switches to occur only in a selected subset of tiles and inhibiting statement reordering in other selected ones. The set of possible program variants defined by a tiling covers all possible interleavings of the concurrent programs. We can analyze these program variants in parallel on a cluster using any off-the-shelf backend tool. We implement the approach building on the CSeq framework and use Lazy-CSeq with CBMC backend for the final analysis. We experimentally show that we can find bugs in very hard concurrency benchmarks, `eliminationstack` and `safestack`, under different memory models (SC and PSO) by analysing only a modest number of randomly picked program variants. In comparison to analyzing the original program, our approach reduces time and memory footprint of the backend analysis tool when launched on a program variant. In summary, we are able to reduce the wall clock time to find Heisenbugs by at least two orders of magnitude on the hardest known concurrency benchmarks.

Future Directions: The ideas and approaches proposed in this paper may underpin the development of more effective bug-finding approaches for concurrent programs. We briefly discuss this below.

We have conducted preliminary experiments using a BDD-based model checker designed for concurrent Boolean programs [43], [44] on a version of `safestack` that we have translated by hand. By performing a state space exploration up to four rounds of computation, the BDD size exploded and the analysis was unsuccessful. Repeating the experiment by applying our verification approach with this same tool as backend analyzer, the BDD sizes reduced considerably on all produced program variants, and we have been able to find a bug in less than one minute using only a handful of instances. This suggests that analysis based on BDD can be very valuable when combined with our verification approach, particularly for lock-free implementations of data-structures, and should be further developed to handle multi-threaded C programs.

In our experiments, we use a bounded model checking backend. This proved to be very efficient on instances that actually contain an interleaving that exposes the bug but very slow on the other (bug-free) instances. We think that abstract interpretation has the potential for quickly discharging bug-free instances. This is indeed corroborated by our preliminary experiments where we have used standard abstractions available in abstract interpreters such as ConcurInterProc [45]. We have in fact discovered that with abstract interpretation we can deem bug-free a significant number of instances orders-of-magnitude faster than BMC.

Effective analysis tools based on sound approximation, such as abstract interpretation, can also play a significant role in building verification approaches for finding extremely rare bugs. For example, in our experiments we are not able to find the bug for `safestack`-TSO although we know that there must be an offending execution. We intend to tackle the problem of finding such extremely rare bugs by combining tools based on sound approximations with a recursive verification approach in the style of a divide-and-conquer algorithm. At each level of the recursion, we split the problem using the tiling as shown in this paper. Then, we run in parallel on each instance two tools: a bug finder and bug-free prover. We also give a timeout to halt them. Now, as soon as one of the two tools succeeds we either report the bug or discard the instance (bottom of the recursion). The method recurs on all the instances where the timeout is reached. This method has the potential to take advantage of both the best available technologies for finding bugs, such as those based on BMC [17], [46], [47], [48], [49] and testing [38], [36], [22], and for proving absence of bugs, such as abstract interpretation [50], enhanced with the VeriSmart approach.

Acknowledgment

The authors acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

REFERENCES

[1] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Trans. Software Eng.*, vol. 37, no. 6, pp. 845–857, 2011.

[2] U. Stern and D. L. Dill, "Parallelizing the mur*phi* verifier," in *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254. Springer, 1997, pp. 256–278.

[3] K. Ohmura and K. Ueda, "c-sat: A parallel SAT solver for clusters," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 524–537.

[4] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE Computer Society, 2008, pp. 1–6.

[5] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, 2008.

[6] J. Schumann, "Sicotheo: Simple competitive parallel theorem provers," in *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, ser. Lecture Notes in Computer Science, M. A. McRobbie and J. K. Slaney, Eds., vol. 1104. Springer, 1996, pp. 240–244.

[7] A. Wolf and R. Letz, "Strategy parallelism in automated theorem proving," *IJPRAI*, vol. 13, no. 2, pp. 219–245, 1999.

[8] G. Sutcliffe and D. Seyfang, "Smart selective competition parallelism ATP," in *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference, May 1-5, 1999, Orlando, Florida, USA*, A. N. Kumar and I. Russell, Eds. AAAI Press, 1999, pp. 341–345.

[9] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," 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. 446–455.

[10] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato, "Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 807–812.

[11] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded C programs via lazy sequentialization," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 585–602.

[12] D. Vyukov, "Bug with a context switch bound 5," 2010, https://social.msdn.microsoft.com/Forums/en-US/91c1971c-519f-4ad2-816d-149e6b2fd916/bug-with-a-context-switch-bound-5?forum=chess.

[13] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, P. B. Gibbons and M. Adler, Eds. ACM, 2004, pp. 206–215.

[14] E. M. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and verilog programs using bounded model checking," in *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*. ACM, 2003, pp. 368–371.

[15] M. Müller-Olm, *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 3800.

[16] B. Fischer, O. Inverso, and G. Parlato, "Cseq: A concurrency pre-processor for sequential C verification tools," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 710–713.

[17] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391.

[18] 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, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[19] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.

[20] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[21] E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato, "Lazy sequentialization for TSO and PSO via shared memory abstractions," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 193–200.

[22] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *TOPC*, vol. 2, no. 4, pp. 23:1–23:37, 2016.

[23] D. Detlefs, C. H. Flood, A. Garthwaite, P. A. Martin, N. Shavit, and G. L. S. Jr., "Even better dcas-based concurrent deques," in *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings*, ser. Lecture Notes in Computer Science, M. Herlihy, Ed., vol. 1914. Springer, 2000, pp. 59–73.

[24] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza, "Tractable refinement checking for concurrent objects," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds. ACM, 2015, pp. 651–662.

[25] J. Barnat, L. Brim, and I. Cerná, "Cluster-based LTL model checking of large systems," in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 259–279.

[26] J. Barnat, L. Brim, and P. Rockai, "Scalable multi-core LTL model-checking," in *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, ser. Lecture Notes in Computer Science, D. Bosnacki and S. Edelkamp, Eds., vol. 4595. Springer, 2007, pp. 187–203.

[27] G. J. Holzmann and D. Bosnacki, "The design of a multicore extension of the SPIN model checker," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 659–674, 2007.

[28] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura, "A concurrent portfolio approach to SMT solving," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 715–720.

[29] G. J. Holzmann, "Cloud-based verification of concurrent software," in *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, ser. Lecture Notes in Computer Science, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer, 2016, pp. 311–327.

[30] S. Qadeer and D. Wu, "KISS: keep it simple and sequential," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, W. Pugh and C. Chambers, Eds. ACM, 2004, pp. 14–24.

[31] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, 2009.

[32] S. La Torre, P. Madhusudan, and G. Parlato, "Reducing context-bounded concurrent reachability to sequential reachability," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 477–492.

[33] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Lazy-cseq: A lazy sequentialization tool for C - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 398–401.

[34] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato, "Concurrent program verification with lazy sequentialization and interval analysis," in *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, ser. Lecture Notes in Computer Science, A. E. Abbadi and B. Garbinato, Eds., vol. 10299, 2017, pp. 255–271.

[35] ——, "Lazy sequentialization for the safety verification of unbounded concurrent programs," in *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, ser. Lecture Notes in Computer Science, C. Artho, A. Legay, and D. Peled, Eds., vol. 9938, 2016, pp. 174–191.

[36] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 37–47.

[37] N. Razavi, A. Farzan, and A. Holzer, "Bounded-interference sequentialization for testing concurrent programs," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 7609. Springer, 2012, pp. 372–387.

[38] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 267–280.

[39] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding: an empirical study," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, J. E. Moreira and J. R. Larus, Eds. ACM, 2014, pp. 15–28.

[40] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer, "Preemption sealing for efficient concurrency testing," in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 420–434.

[41] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps, "Conseq: detecting concurrency bugs through sequential errors," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 251–264.

[42] N. Razavi, F. Ivancic, V. Kahlon, and A. Gupta, "Concurrent test generation using concolic multi-trace analysis," in *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and A. Igarashi, Eds., vol. 7705. Springer, 2012, pp. 239–255.

[43] S. La Torre, P. Madhusudan, and G. Parlato, "Analyzing recursive programs using a fixed-point calculus," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 211–222.

[44] ——, "Model-checking parameterized concurrent programs using linear interfaces," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. B. Jackson, Eds., vol. 6174. Springer, 2010, pp. 629–644.

[45] B. Jeannet, "Relational interprocedural verification of concurrent programs," in *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, D. V. Hung and P. Krishnan, Eds. IEEE Computer Society, 2009, pp. 83–92.

[46] T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato, "Lazy-cseq 2.0: Combining lazy sequentialization with abstract interpretation - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 375–379.

[47] E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato, "Mu-cseq 0.4: Individual memory location unwindings - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, M. Chechik and J. Raskin, Eds., vol. 9636. Springer, 2016, pp. 938–941.

[48] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato, "Verifying concurrent programs by memory unwinding," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 551–565.

[49] E. Tomasco, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato, "Using shared memory abstractions to design eager sequentializations for weak memory models," in *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*, ser. Lecture Notes in Computer Science, A. Cimatti and M. Sirjani, Eds., vol. 10469. Springer, 2017, pp. 185–202.

[50] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252.