Inspect: A Runtime Model Checker for Multithreaded C Programs

Yu Yang
Xiaofang Chen
Ganesh Gopalakrishnan

UUCS-08-004

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

Abstract

We present `Inspect`, a runtime model checker for revealing concurrency bugs in multithreaded C programs. `Inspect` instruments a given program at all global interaction points, and with the help of a new scheduler, examines all relevant thread interleavings under dynamic partial order reduction (DPOR). While the ideas behind `Inspect` are well known, there hasn't been a previously reported effort in which these ideas are applied to multithreaded C programs. We report on our engineering efforts to endow `Inspect` with (i) automatic source program instrumentation, (ii) practical DPOR implementation, and (iii) optimizations such as using locksets to compute more precise co-enabled relation. Our initial experience shows that such a tool can, indeed, be very effective for obtaining a handle on the notorious complexity of thread programming.

1 Introduction

Writing correct multithreaded programs is difficult. Many "unexpected" thread interactions can only be manifested with intricate low-probability event sequences. As a result, they often escape conventional testing, and manifest years after code deployment [1]. Many tools have been designed to address this problem. They can be generally classified into three categories: dynamic detection, static analysis, and model checking.

Eraser [2] and Helgrind [3] are two examples of data race detectors that dynamically track the set of locks held by shared objects during program execution. These tools predict potential data races by inferring them based on the observation of one feasible execution path. There is no guarantee that the program is free from data races if no error is reported (full coverage is not guaranteed).

Tools such as RacerX [4], ESC/Java [5], and LockSmith [6] detect potential errors in programs by statically analyzing the source code. Since they do not get the benefit of analyzing concrete executions, the false positive (false alarm) rates of these tools can be high.

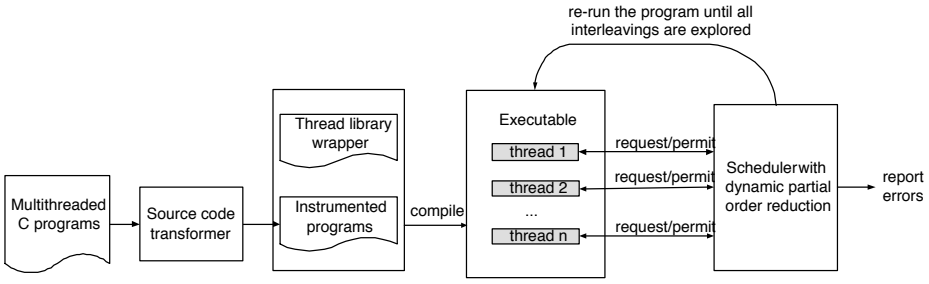

Figure 1: `Inspect`'s workflow

Traditional model checking can guarantee complete coverage, but on implicitly or explicitly extracted models before model checking (*e.g.*, [7, 8, 9]), or in the context of languages whose interpreters can be easily modified for backtracking (*e.g.*, [10]). As far as we know, none of these model checkers can easily check (or be easily adapted to check) general application-level multithreaded C programs. For instance, if we want to follow Java PathFinder's [10] approach to check multithreaded C programs, we will have to build a virtual machine that can handle C programs. This is very involved. For model checkers like Bogor [7], Spin [9], Zing [8] and so on, modeling library functions and the runtime environment of C programs is difficult as well as error-prone: the gap between modeling languages and programming languages is unbridgeably large in many cases.

Predicate abstraction based model checkers, such as [11, 12, 13], have demonstrated the

abilities to prove properties of sequential programs. However, these model checkers haven't been able to verify realistic multithreaded programs because realistic programs often involve a lot of complicated data structures, and existing decision procedures haven't been able to handle them yet.

Verisoft [14] is able to check concurrent C/C++ programs without incurring modeling overheads. However, Verisoft focuses on message-passing based concurrent programs that interact only through inter-process communication mechanisms. In a multithreaded program, the threads can affect each other not only through explicit synchronization/mutual exclusion primitives, but also through read/write operations on shared data objects. We need a mechanism that is different from Verisoft to take the control of scheduling away from the operating system and explore different interleavings.

1.1 Our Contribution

We designed `Inspect`, a runtime model checker for data races, deadlocks, or other concurrency related errors in multithreaded C programs for a fixed environment (fixed set of input drivers). `Inspect` can systematically explore all possible interleavings of a multithreaded C program under a specific input driver, and guarantee that there are no concurrency errors. `Inspect` does not generate false positives, and to our best knowledge, is the first runtime model checker that can handle multithreaded C programs.

An overview of `Inspect` is shown in Figure 1. It consists of three parts:

- A source code transformer to instrument the program at the source code level.
- A thread library wrapper that helps intercept the thread library calls
- A centralized scheduler that schedules the interleaved executions of the threads.

Given a multithreaded program, `Inspect` first instruments the program with code that is used to communicate with the scheduler. Thereafter it compiles the program into an executable and runs the executable repeatedly under the control of the scheduler until all relevant interleavings among the threads are explored. Before performing any operation that might have side effects on other threads, the instrumented program sends a request to the scheduler. The scheduler can block the requester by postponing a reply. We use blocking sockets as communication channels between the threads and the scheduler. As the number of possible interleavings grows exponentially with the size of the program,

we implemented the dynamic partial order reduction (DPOR [15]) algorithm to reduce the search space. [1]

2 Runtime Model Checking

Model checking is a technique for verifying a transition system by exploring its state space. Cycles in the state space are detected by checking whether a state has been visited before or not. Usually the visited states information is stored in a hash table. Runtime model checkers explore the state space by executing the program concretely and observing its visible operations. Runtime model checkers do not keep the search history because it is not easy to capture and restore the state of a program which runs concretely. As a result, runtime model checkers avoid the memory blowup problem that many static model checkers suffer. As a trade-off, runtime model checkers are not capable of checking programs that have cyclic state spaces.

`Inspect` follows the common design principles of a runtime model checker, and uses a depth-first strategy to explore the state space. As a result, `Inspect` can only handle programs that can terminate in a finite number of steps. Fortunately the execution of many multithreaded programs terminates eventually.[2]

3 An Example

In this section we show how `Inspect` works with a simple example which captures a common concurrent scenario in database systems. Suppose that a shared database supports two distinct classes of operations, A and B. Let the semantics of the two types of operations be such that multiple operations of the same class can run concurrently, but operations belonging to different classes cannot be run concurrently. Figure 2 is a buggy implementation which can lead to a deadlock. Global variables `A_count` and `B_count` capture the number of threads that are performing operations A and B respectively. Here, the variable `lock` is used for mutual exclusion between threads, and `mutex` is used to guarantee the atomicity of updating the counters, `a_count` and `b_count`.

[1] The method we present in this paper can also be applied to C++ programs. However, due of the lack of a C++ front end for source code transformation, we need manual instrumentation for C++ programs.

[2] If termination is not guaranteed, `Inspect` can still work by depth-bounding the search.

shared variables among threads:

```
pthread_mutex_t mutex, lock;
int A_count = 0, B_count = 0;
```

class A operation:

```
 1:   pthread_mutex_lock(&mutex);
 2:   A_count++;
 3:   if (A_count == 1) {
 4:      pthred_mutex_lock(&lock);
 5:   }
 6:   pthread_mutex_unlock(&mutex);
 7:   performing class A operation;
 8:   pthread_mutex_lock(&mutex);
 9:   A_count--;
10:   if (A_count == 0){
11:      pthread_mutex_unlock(&lock);
12:   }
13:   pthread_mutex_unlock(&mutex);
```

class B operation:

```
1:   pthread_mutex_lock(&mutex);
2:   B_count++;
3:   if (B_count == 1){
4:     pthred_mutex_lock(&lock);
5:   }
6:   pthread_mutex_unlock(&mutex);
7:   performing class B operation;
8:   pthread_mutex_lock(&mutex);
9:   B_count--;
10:  if (B_count == 0){
11:     pthread_mutex_unlock(&lock);
12:  }
13:  pthread_mutex_unlock(&mutex);
```

Figure 2: An example on concurrent operations in a shared database

Conventional testing might miss the potential deadlock hidden in the code as it runs with random scheduling. In general it is difficult to get a specific scheduling that leads to the error. `Inspect` first instruments the program with code that can take the control of scheduling away from the runtime system. Then it compiles the instrumented code into an executable. With the help of a centralized scheduler, `Inspect` can systematically explore

relevant interleavings. As for this example, `Inspect` will report that $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow b_1 \rightarrow b_2$ is an interleaving which leads to a deadlock (here a_i and b_i stand for line i of class A and class B threads).

4 Inspect

4.1 Identifying Threads and Shared Objects Out of Multiple Runs

When `Inspect` re-executes the program under test, the runtime environment may change across re-executions. For instance, the threads may not be allocated to the same identity by the operating system. Also, dynamically-created shared objects (e.g., mallocs) may not reside in the same physical memory address in different runs. Handling these practical issues is essential for the success of runtime model checking.

We handle these issues in `Inspect` aided by a few (practically realistic) assumptions. First, we assumes that given the same external input, multiple threads in a program are always created in the same order across different runs. Banking on this fact, we can identify threads (which may be allocated different thread IDs in various re-executions) across two different runs by examining the sequence of thread creations. We let each thread register itself to the scheduler. If the threads are created in the same sequential order in different runs, threads will be registered in the same order. In this way, we can easily assign the same ID for the same thread across multiple runs.

Identifying the shared objects is done in the same manner: if two runs of the program have the same visible operation sequence, the shared objects will also be created with malloc, etc., in the same sequence. As a result, shared objects across multiple runs can be identified, even though the actual objects may be getting created at different memory addresses in different re-executions.

4.2 Instrumenting the code

We describe how `Inspect` works for a simple C-like language shown in Figure 3. A program is composed of a *main* function and a set of threads. Each thread is a sequence of statements. The condition expression in the *if* statement is simplified as a variable. `Inspect` uses CIL [16] to convert more complex statements into this simplified form by

$$
\begin{array}{rcl}
P & ::= & T^* \\
T & ::= & main \mid thread \\
thread & ::= & Stmt^* \\
main & ::= & Stmt^* \\
Stmt & ::= & [l:]s \\
s & ::= & lhs \leftarrow e \mid if\ lhs\ then\ goto\ l' \mid create \mid join \mid exit \mid lock \mid unlock \mid lhs \leftarrow malloc \\
lhs & ::= & v \mid \&v \mid *v \\
e & ::= & lhs \mid lhs\ \ op\ \ lhs \\
 & & where\ op \in \{+, -, *, /, \%, <, >, \leq, \geq, \neq, =, ...\}
\end{array}
$$

Figure 3: Syntax of a simple language that is similar to C

introducing temporary variables

Before instrumentation	After instrumentation
create	inspect_create()
join	inspect_join()
exit	inspect_exit()
lock	inspect_lock()
unlock	inspect_unlock()
thread	thread_begin thread thread_end
main	global_object_registration() thread_begin main thread_end
//allocating memory for a shared object lhs ← malloc	lhs ← malloc() object_registration(&lhs)
//*if* lhs *is a shared object* lhs ← v	write_shared(&lhs, v)
//*if* v *is a shared object* lhs ← v;	lhs ← read_shared(&v)

Figure 4: `Inspect`'s instrumentation

Figure 4 shows how `Inspect` instruments a multithreaded program based on the syntax shown in Figure 3. The instrumentation does the following things:

- Replace the function calls to the thread library routines with function calls to `Inspect` wrapper functions.
- Add extra code before thread starting/exiting points to notify the scheduler.
- Add object registration code at the beginning of *main* function for global objects. Object registration code is also needed after *malloc* operations which allocate new objects which are shared among threads.
- For each read/write access on data objects that are shared among threads, `Inspect` intercepts the operations by adding a wrapper around it.

To achieve the last step, we need to know whether an update to a data object is a visible operation or not. Doing this exactly is undecidable,as it amounts to context sensitive language reachability [17]. We conservatively over-approximate this step by performing an inter-procedural flow-sensitive alias analysis [18] on the program. Based on the results of the alias analysis, we have a may-escape analysis [19] on the program to find all operations on shared objects that may "escape" the thread scope. As the may-escape analysis is an over-approximation of all-possible shared variables among threads, our instrumentation is safe for intercepting all the visible operations in the concrete execution. Figure 5 shows the instrumented code of class A thread in Figure 2. It is the engineering of these details that sets `Inspect` apart from previous prototype implementations of DPOR, and for the first time makes it possible to assess the impact of a well-engineered DPOR algorithm in the setting of realistic multithreaded C programs.

4.3 Dynamic Partial Order Reduction

While the number of possible interleavings grows exponentially as the program becomes large, we use dynamic partial order reduction [15] to avoid exploring redundant interleavings. In this section, we describe our implementation of the DPOR algorithm, and our improvement on the original.

4.3.1 Definitions

Partial order reduction (POR) techniques [20] are those that avoid interleaving independent transitions during search. Given the set of enabled transitions from a state s, partial order reduction algorithms try to explore only a (proper) subset of the enabled transitions at s,

```
void *thread_A(void *arg )
{
  void *__retres2 ;
  int __cil_tmp3 ;
  int __cil_tmp4 ;
  int  __cil_tmp5 ;
  int  __cil_tmp6 ;
  int  __cil_tmp7 ;
  int  __cil_tmp8 ;
  int  __cil_tmp9 ;
  int  __cil_tmp10 ;

  inspect_thread_start("thread_A");
  inspect_mutex_lock(& mutex);
  __cil_tmp7 = read_shared_0(& A_count);
  __cil_tmp3 = __cil_tmp7 + 1;
  write_shared_1(& A_count, __cil_tmp3);
  __cil_tmp8 = read_shared_2(& A_count);
  __cil_tmp4 = __cil_tmp8 == 1;
  if (__cil_tmp4) {
    inspect_mutex_lock(& lock);
  }
  inspect_mutex_unlock(& mutex);
  inspect_mutex_lock(& mutex);
  __cil_tmp9 = read_shared_3(& A_count);
  __cil_tmp5 = __cil_tmp9 - 1;
  write_shared_4(& A_count, __cil_tmp5);
  __cil_tmp10 = read_shared_5(& A_count);
  __cil_tmp6 = __cil_tmp10 == 0;
  if (__cil_tmp6) {
    inspect_mutex_unlock(& lock);
  }
  inspect_mutex_unlock(& mutex);
  __retres2 = (void *)0;
  inspect_thread_end();
  return (__retres2);
}
```

Figure 5: Instrumented code for the class A thread shown in Figure 2.

and at the same time guarantee that the properties of interest will be preserved. Such a subset is called *persistent set*.

Static POR algorithms compute the persistent set of a state immediately after reaching it. In the context of multithreaded C/C++ programs, persistent sets computed statically will be excessively large because of the limitations of static analysis. For instance, if two transitions leading out of s access an array `a[]` by indexing it at locations captured by expressions `e1` and `e2` (i.e., `a[e1]` and `a[e2]`), a static analyzer may not be able to decide whether `e1=e2`. Flanagan and Godefroid introduced dynamic partial-order reduction (DPOR) [15] to dynamically compute smaller persistent sets (smaller persistent sets are almost always better).

Given a state s and a transition t, we use the following notations:

- $t.tid$ denotes the identity of the thread that executes t.
- $next(s, t)$ refers to the state which is reached from s by executing t.
- $s.enabled$ denotes the set of transitions that are enabled from s. A thread p is enabled in a state s if there exists some transition t such that $t \in s.enabled$ and $t.tid = p$.
- $s.backtrack$ refers to the backtrack set at state s. $s.backtrack$ is a set of thread identities. Here, $\{t \mid t.tid \in s.backtrack\}$ is the set of transitions which are enabled but have not been executed from s.
- $s.done$ denotes the set of threads examined at s. Similar to $s.backtrack$, $s.done$ is also a set of thread identities. Here, $\{t \mid t.tid \in s.done\}$ is the set of transitions that have been executed from s.
- $s.sleep$ refers to the sleep set associated with a state. A sleep set [21] is a set of transitions that are enabled but will not be executed from a state s. We use sleep sets in our implementation for further reducing redundant interleavings.

In DPOR, given a state s, the persistent set of s is not computed immediately after reaching s. Instead, DPOR explores the states that can be reached from s with depth-first search, and dynamically computes the persistent set of s. Assume $t \in s.enabled$ is the transition which the model checker chose to execute, and t' is a transition that can be enabled with DFS (with one or more steps) from s by executing t. For each to-be-executed transition t', DPOR will check whether t' and t are dependent and can be enabled concurrently (i.e. *co-enabled*). If t' and t are dependent and can be co-enabled, $t'.tid$ will be added to the $s.backtrack$. Later, when backtracking during DFS, if a state s is found with non-empty $s.backtrack$, DPOR

```
1:  StateStack S;
2:  State s, s';

3:  DPOR() {
4:      run P, which is the program under test;
5:      s ← the initial state of the program;
6:      while (s.enabled ≠ ∅) {
7:          S.push(s);
8:          choose t ∈ s.enabled;
9:          s ← next(s, t);
10:         update_backtrack_info(s); //described in Figure 7
11:     }
12:     while (¬S.empty()) {
13:         s ← S.pop();
14:         if (s.backtrack ≠ ∅)
15:           backtrack_checking(s);
16:     }
17: }

18: backtrack_checking(State s_bt) {
19:     replay the program from the beginning until s_bt
20:     s ← s_bt
21:     choose t, t ∈ s.enabled ∧ t.tid ∈ s.backtrack
22:     s.backtrack ← s.backtrack \ {t.tid};
23:     s.sleep ← {t ∈ s.enabled | t.tid ∈ s.done}
24:     s.done ← s.done ∪ {t.tid};
25:     repeat
26:       S.push(s);
27:       s' ← next(s, t);
28:       s'.sleep ← { t' ∈ s.sleep | (t, t') are independent};
29:       s'.enabled ← s'.enabled \ s'.sleep;
30:       s ← s';
31:       update_backtrack_info();
32:       choose t ∈ s.enabled;
33:     until (s.enabled = ∅)
34: }
```

Figure 6: Our implementation of dynamic partial-order reduction

will pick one transition t such that $t \in s.enabled$ and $t.tid \in s.backtrack$, and explore a new branch of the state space by executing t.

```
1:  update_backtrack_info(State s) {
2:      let T be the sequence of transitions that are executed from the initial state of the
            program to reach state s;
3:      for each thread h {
4:          let t_n ∈ s.enabled, t_n.tid = h;
5:          let t_d be the latest transition in T that is dependent and may be co-enabled with
                t_n;
6:          if (t_d ≠ null) {
7:              let s_d be the state in the state stack S from which t_d is executed;
8:              let E be {q ∈ s_d.enabled | q.tid = h, or q in T, q happened after t_d and
                    is dependent with some transition in T which was executed by h and
                    happened after q }
9:              if (E ≠ ∅)
10:                 choose any q in E, add q.tid to s_d.backtrack;
11:             else
12:                 s_d.backtrack ← s_d.backtrack ∪ {q.tid | q ∈ s_d.enabled};
13:         }
14:     }
15: }
```

Figure 7: Update the backtracking information for the states in the search stack

Figure 6 shows our implementation of the DPOR algorithm. Given a program under test, DPOR first runs the program randomly(Line 6-11). After this, DPOR keeps backtrack until the state stack is empty (Line 12-16). The key point is that in this process, each time a new state is reached, `update_backtrack_info` is called to update the backtrack sets for states in the search stack (Line 10 and line 31).

The detail of `update_backtrack_info` is explained in Figure 7. The backtrack set of a state, $s_d.backtrack$, is updated while exploring a state s reached from $s_d.backtrack$ under DFS. Observe from line 10 of Figure 7 that we add to $s_d.backtrack$ a thread id q, where s_d is the most recent state, searching back from s, where a transition that depends on a transition t_n that is about to be taken from s occurs. When the DFS unwinds to state $s_d.backtrack$, the backtrack set is consulted and the threads recorded in there are scheduled.

4.3.2 An Example

Here we give out an example to illusrate the DPOR algorithm. Consider two threads t_1 and t_2 which share two variables `x` and `y`:

$$\begin{array}{ll} t_1: & x = 1; \quad x = 2 \\ t_2: & y = 1; \quad x = 3 \end{array}$$

Assume the first random execution of the program is:

$$t_1 : x = 1; \;\; t_1 : x = 2; \;\; t_2 : y = 1; \;\; t_2 : x = 3;$$

Before executing the last transition $t_2 : x = 3$, DPOR will add a backtracking point for thread t_2 before the last transition of t_1. With sleep sets enabled, DPOR will force the following execution (the sleeps sets are shown in the parenthesis) :

$$(\varnothing)t_1 : x = 1; \;\; (\{t_1 : x = 2\})t_2 : y = 1; \;\; (\{t_1 : x = 2\})t_2 : x = 3; \;\; (\varnothing)t_1 : x = 2;$$

which in turn forces

$$(\{t_1 : x = 1\})t_2 : y = 1; \;\; (\{t_1 : x = 1\})t_2 : x = 3; \;\; (\varnothing)t_1 : x = 1; \;\; (\varnothing)t_1 : x = 2;$$

4.3.3 Avoiding Redundant Backtracking

One optimization we made on DPOR algorithm is on avoiding redundant backtracking. State backtracking is an expensive operation for runtime model checkers as the model checker need to restart the program, and replay the program from the initial state until the backtrack point. Obviously, we want to avoid backtracking as much as possible to improve efficiency. Line 4 of Figure 7 is the place in DPOR where a backtrack point is identified. It treats t_d, which is dependent and may-be co-enabled with t_n as a backtrack point. However, *if two transitions that may be co-enabled are never co-enabled, we may end up exploring redundant backtrackings, and reduce the efficiency of DPOR*.

We use locksets to avoid exploring transition pairs that can not be co-enabled. Each transition t is associated with the set of locks that are held by the thread which executes t. With these locks, we compute the may co-enabled relation more precisely by testing whether the intersection of the locksets held by two threads is empty or not.

5 Implementation

`Inspect` is designed in a client/server style. The server side is the scheduler which controls the program's execution. The client side is linked with the program under test to communicate with the scheduler. The client side includes a wrapper for the pthread library, and facilities for communication with the scheduler.

We have the scheduler and the program under test communicate using Unix domain sockets. Comparing with Internet domain sockets, Unix domain sockets are more efficient as they do not have the protocol processing overhead, such as adding or removing the network headers, calculating the check sums, sending the acknowledgments, etc. Besides, the Unix domain datagram service is reliable. Messages will neither get lost nor delivered out of order. we use CIL [16] to implement the program analysis and transformation part. An initial version of `Inspect` is available at http://www.cs.utah.edu/∼yuyang/inspect.

In *pfscan*, `Inspect` found an error that a condition variable is used without initialization. This may mess up synchronization among threads and end up with incorrect results. In addition, two mutexes that are initialized at the beginning of the program never get released, which may result in resource leakage.

As for the other two benchmarks, we did not find any bugs. However, when we deliberately inserted some races/deadlock bugs into these benchmarks, `Inspect` found all of them.

6 Additional Related Work

CMC [22] verifies C/C++ programs by using a user-model Linux as a virtual machine. CMC captures the virtual machine's state as the state of a program. Unfortunately, CMC is not fully-automated. As CMC takes the whole kernel plus the user space as the state, it is not convenient for CMC to adapt the dynamic partial order reduction method.

ConTest [23] debugs multithreaded programs by injecting context switching code to randomly choose the threads to be executed. As randomness does not guarantee all interleavings will be explored for a certain input, it is possible that ConTest can miss bugs. Given an input, if there exists some interleavings that can lead to error, `Inspect` can guarantee that the error will be caught.

Lei et al. [24] designed RichTest, which used reachability testing to detect data races in

concurrent programs. Reachability testing views an execution of a concurrent program as a partially-ordered synchronization sequence. However, RichTest does not address the practical issues like how to control the scheduling of multithreaded C program which `Inspect` solves.

Helmstetter et al. [25] show how to generate scheduling based on dynamic partial order reduction. It solves the problem of exploring different interleavings in the context of SystemC models. Inspect handles general multithreaded C applications, using a different method to handle the practical scheduling problem.

CHESS [26] is a runtime model checker for testing multithreaded programs. CHESS assumes that a program is data-race free, every access to shared objects is protected by some locks. `Inspect` does not require this assumption. CHESS takes control of the scheduling by intercepting only library calls. Inspect intercepts not only library calls, but also visible operations on data objects by performing source code transformation, as discussed in Section 4.2. As a result of this, CHESS captures the happen-before relation of the events, and reports a race when it observes two events between which there is no happen-before relation. Different from CHESS, `Inspect` reports a race if and only if a real racing scenario is observed.

7 Conclusion

In summary, in this paper, we present a practical runtime model checker for checking concurrency-related errors in multithreaded C programs. Our method works by automatically enumerating all possible interleavings of the threads in a multithreaded program, and forcing these interleavings to execute one by one. We use dynamic partial-order reduction to eliminate unnecessary explorations. Our preliminary results show that this method is promising for revealing bugs in real multithreaded C programs.

In the future, `Inspect` can be improved by combining the static analysis techniques with the dynamic partial order reduction to further reduce the number of interleavings we need to explore to reveal errors. `Inspect` can also adapt more efficient algorithms such as Goldilocks [27] for computing happen-before relations to improve efficiency. We can also improve the automated instrumentation part by employing more efficient and precise pointer-alias analysis so as to reduce the communication overhead between the scheduler and the program under test.

References

[1] Edward A. Lee. The problem with threads. volume 39, pages 33–42, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[2] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[3] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[4] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[5] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[6] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[7] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276, 2003.

[8] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.

[9] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[10] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *ASE*, pages 3–12, 2000.

[11] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[12] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.

[13] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[14] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[15] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 110–121. ACM, 2005.

[16] http://manju.cs.berkeley.edu/cil/.

[17] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2006.

[18] K. D. Cooper and K. Kennedy. Fast interprocedual alias analysis. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–59, New York, NY, USA, 1989. ACM.

[19] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23, New York, NY, USA, 2001. ACM Press.

[20] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[21] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[22] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI*, 2002.

[23] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[24] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006.

[25] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. *fmcad*, 0:171–178, 2006.

[26] http://research.microsoft.com/projects/CHESS/.

[27] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Formal Approaches to Software Testing and Runtime Verification, LNCS*, pages 193–208, Berlin, Germany, 2006. Springer.