Runtime Model Checking of Multithreaded C/C++ Programs

Yu Yang Xiaofang Chen Ganesh Gopalakrishnan Robert M. Kirby

School of Computing, University of Utah
Salt Lake City, UT 84112, U.S.A.

Abstract. We present `inspect`, a tool for model checking safety properties of multithreaded C/C++ programs where threads interact through shared variables and synchronization primitives. The given program is mechanically transformed into an instrumented version that yields control to a centralized scheduler around each such interaction. The scheduler first enables an arbitrary execution. It then explores alternative interleavings of the program. It avoids redundancy exploration through dynamic partial order reduction(DPOR) [1]. Our initial experience shows that `inspect` is effective in testing and debugging multithreaded C/C++ programs. With `inspect`, we have been able to find many bugs in real applications.

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. 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 try to detect potential data races by inferring them based on one feasible execution path. There is no guarantee that the program is free from data races if no error is reported (i.e., 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 warning rates of these tools can be high. They also provide no guarantee of full coverage.

Traditional model checking can guarantee complete coverage, but on implicitly or explicitly extracted models before model checking (*e.g.*, [7,8,9,10,11]), or in the context of languages whose interpreters can be easily modified for backtracking (*e.g.*, [12]). As far as we know, none of these model checkers can easily check (or be easily adapted to check) general application-level multithreaded C/C++ programs. For instance, if we want to follow Java PathFinder's [12] approach to check multithreaded C/C++ programs, we will have to build a virtual

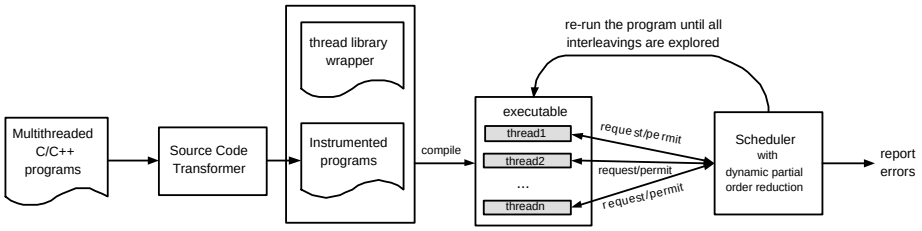

Fig. 1. Inspect's workflow

machine that can handle C/C++ programs. This is very involved. For model checkers like Bogor[8], Spin[11], Zing [10] and so on, modeling library functions and the runtime environment of C/C++ programs is very involved as well as error-prone: the gap between modeling languages and programming languages is unbridgeably large in many cases.

To the best of the authors' knowledge, Verisoft [13] is the only model checker that is able to check concurrent C/C++ programs without incurring modeling overheads. Unfortunately, Verisoft focuses on concurrent programs that interact only through inter-process communication mechanisms. In a real-world 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.

To address these problems, we designed `inspect`, a runtime model checker for systematically exploring all possible interleavings of a multithreaded C/C++ program under a specific testing scenario. In other words, the reactive program under test is closed by providing a test driver, and `inspect` examines *all* thread interleavings under this driver.

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, and 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. Then `inspect` 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 an adaption of the dynamic partial order reduction (DPOR [1]) algorithm proposed by Flanagan and Godefroid to reduce the search space.

`Inspect` can check application-level C/C++ programs that use POSIX threads[14]. `Inspect` supports not only mutual exclusive lock/unlock, but operations on con-

dition variables, including wait, signal and broadcast. The errors that `inspect` can detect include data races, deadlocks, and incorrect usages of thread library routines. When an error is located, `inspect` reports the error along with the trace that leads to the error, which facilities debugging. The key features of our work, and some of the challenges we overcame are as follows:

- We design `inspect`, an *in situ* runtime model checker, that can efficiently check multithreaded C/C++ programs. `Inspect` not only supports mutual exclusive locks, but also wait/signal, and read/write locks. The ability to model check programs containing these constructs makes `inspect` a unique tool.
- We have evaluated `inspect` on a set of benchmarks and confirmed its efficacy in detecting bugs.
- We have designed and implemented an algorithm to automate the source code instrumentation for runtime model checking.
- Since `inspect` employs stateless search, it relies on re-execution to pursue alternate interleavings. However, during re-execution, the runtime environment of the threaded code can change. This can result in the threads not being allotted the same id by the operating system. Also, dynamically-created shared objects may not reside in the same physical memory address in different runs. We have suitably addressed these challenges in our work. We also employ lock-sets, and additionally sleep-sets (the latter is recommended in [1]), to eliminate redundant backtrack points during DPOR.

2 Background

2.1 Multithreaded Programs in C/C++

Threading is not part of the C/C++ language specification. Instead, it is supported by add-on libraries. Among many implementations of threading, POSIX threads [14] are perhaps the most widely used.

Mutex and *condition variable* are two common data structures for communication between threads. Mutexes are used to give threads exclusive access to critical sections. Condition variables are used for synchronization between threads. Each condition variable is always used together with an associated mutex. When a thread requires a particular condition to be true before it can proceed, it waits on the associated condition variable. By waiting, it gives up the lock and blocks itself. The operations of releasing the lock and blocking the thread should be atomic. Any thread that subsequently causes the condition to be true may then use the condition variable to notify a thread waiting for the condition. A thread that has been notified regains the lock and can then proceed.

The POSIX thread library also provides *read-write locks* and *barriers*. A read-write lock allows concurrent read access to an object but requires exclusive access for write operations. Barrier provides explicit synchronization for a set of threads. As barriers seem not to be frequently used in multithreaded programs (we did not encounter any uses, except in some tutorials), we do not consider them here.

2.2 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 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. [1]

2.3 Dynamic Partial Order Reduction

Partial order reduction (POR) techniques[15] 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, 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 s immediately after reaching it. In our context, 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) [1] to dynamically compute smaller persistent sets (smaller persistent sets are almost always better).

In the rest of the paper, 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. $\{t \mid t.tid \in s.backtrack\}$ 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. $\{t \mid t.tid \in s.done\}$ is the set of transitions that have been executed from s.

```
1:  StateStack S;
2:  TransitionSequence T;

3:  DPOR( ) {
4:      let s be the top state on stack S;
5:      update_backtrack_info(s);
6:      if (∃ thread p, ∃t ∈ s.enabled, t.tid = p) {
7:          s.backtrack = {p};
8:          s.done = ∅;
9:          while (∃q ∈ s.backtrack \ s.done) {
10:             s.done = s.done ∪ {q};
11:             s.backtrack = s.backtrack \ {q};
12:             let t_n ∈ s.enabled, t_n.tid = q;
13:             T.append(t_n);
14:             S.push(next(s, t_n));
15:             DPOR();
16:             T.pop_back();
17:             S.pop();
18:         }
19:     }
20: }

21: update_backtrack_info(State s) {
22:     for each thread p {
23:         let t_n ∈ s.enabled, t_n.tid = p;
24:         let t_d be the latest transition in T that is dependent and may be co-enabled
                with t_n;
25:         if (t_d ≠ null) {
26:             let s_d be the state in S from which t_d is executed;
27:             let E be {q ∈ s_d.enabled | q.tid = p, or q in T, q happened after t_d
                    and is dependent with some transition in T which was executed by
                    p and happened after q }
28:             if (E ≠ ∅)
29:                 choose any q in E, add q.tid to s_d.backtrack;
30:             else
31:                 s_d.backtrack = s_d.backtrack ∪ {q.tid | q ∈ s_d.enabled};
32:         }
33:     }
34: }
```

Fig. 2. Dynamic partial-order reduction

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.

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

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 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. Figure 2 recapitulates the DPOR algorithm (this is the same as the one given, as well as proved correct in [1]; we merely simplified some notations).

3 An Example

In this section we consider the following example, which captures a common concurrent scenario in database systems. Suppose that a shared database supports two distinct classes of operations, A and B. The semantics of the two operations allow multiple operations of the same class to run concurrently, but operations belonging to different classes cannot be run concurrently. Figure 3 is a buggy implementation that attempts to solve this problem. This implementation can lead to a deadlock. `a_count` and `b_count` are the number of threads that are performing operations A and B respectively. Here, `lock` is used for the mutual exclusion between threads, and `mutex` is used to guarantee the atomicity of updating the counters, `a_count` and `b_count`.

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 will lead to the error. To systematically explore all possible interleavings, `inspect` needs to take the control of scheduling away from the operating system. We do this by instrumenting the program with code that is used to communicate with a central scheduler. As only visible operations in one thread can have side effects on other threads, we only need to instrument before each visible operation is performed. The instrumented code sends a request to the scheduler. The scheduler can then decide whether the request should be granted permission immediately, or be delayed. The scheduler works as an external observer. In addition, it needs to be notified about the occurrences of the thread start, join, and exit events.

Figure 4 shows the code after instrumentation for threads that perform class A operations. As shown in the figure, each call to the pthread library routines is replaced with a wrapper routine. As `a_count` is a shared variable that multiple threads can access, we insert a read/write request to the scheduler before each access of `a_count`. A call to `inspect_thread_start` is inserted at the entry of the thread to notify that a new thread is started. Similarly, a call to `inspect_thread_end` is inserted at the end of the thread routine.

After compiling the instrumented program, `inspect` obtains an executable that can be run under the central scheduler's control and monitoring. `Inspect` first lets the program run randomly and collects a sequence of visible operations.

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);
```

Fig. 3. An example on concurrent operations in a shared database

This sequence reflects a random interleaving of the threads. If the sequence happens to be an interleaving that can lead to errors, these errors will be reported immediately. Otherwise, `inspect` will try to find a backtrack point out of the trace (as described in Figure 2), and begins monitoring the executable runs, now obtained through another interleaving.

Assume the program shown in Figure 3 has only one class A thread, and one class B thread. In the first run of the instrumented program, `inspect` may observe the visible operation sequence shown in Figure 5, beginning with "(*thread a*)" (which does not contain any errors).

```
inspect_thread_start();
   ...
inspect_mutex_lock(&mutex);
inspect_obj_write( (void*)&a_count );
a_count++;
inspect_obj_read( (void*)&a_count );
if (a_count == 1)
   inspect_mutex_lock(&lock);
inspect_mutex_unlock(&mutex);
  ...
inspect_mutex_lock(&mutex);
inspect_obj_write( (void*)&a_count );
a_count--;
inspect_obj_read( (void*)&a_count );
if (a_count == 0)
   inspect_mutex_unlock(&lock);
inspect_mutex_unlock(&mutex);
  ...
inspect_thread_end();
```

Fig. 4. Instrumented code for class A threads shown in Figure 3

In Figure 5, after observing the random visible operation sequence, `inspect` will, for each visible operation, update the backtrack set for each state in the search stack. In the shown trace, as b_1 and a_6 are both lock acquire operations on shared object `mutex`, and event b_1 is dependent and may be co-enabled with a_6, `inspect` will put the backtracking information after event a_5 (i.e., just before a_6). If `inspect` has explored all the transitions from a state s, it will start backtracking. In the backtracking, if `inspect` reaches a state s where $s.backtrack$ is not empty, `inspect` will execute a transition whose thread id is in $s.backtrack$.

In this example, (i) `inspect` will first re-execute the instrumented program and allow thread a run through $a_1 - a_5$. (ii) now, since the backtrack set contains b, the scheduler will block thread a until thread b has performed the visible operation b_1. Then it will allow thread a and thread b run randomly until all threads ends. In our example, we will observe the alternate sequence of visible operations generated as a result of re-execution, as shown in Figure 6.

After thread b performs the visible operation b_3 as in Figure 6, thread a is trying to acquire `mutex` which is held by thread b; at the same time, thread b is waiting to acquire `lock`, which is held by thread a. *`Inspect` will report this deadlock scenario at this point, and start backtracking.* Now, as there is a newly computed backtrack point at the state from which a_1 is executed, `inspect` will start another backtracking. In general, `inspect` will continue backtracking until there are no backtrack points in the search stack.

$$
\begin{array}{rll}
(\mathit{thread}\ a) \Rightarrow & a_1: & \mathit{acquire}\ \texttt{mutex} \\
& a_2: & \texttt{count\_a}++ \\
& a_3: & \texttt{count\_a} == 1 \\
& a_4: & \mathit{acquire}\ \texttt{lock} \\
& a_5: & \mathit{release}\ \texttt{mutex} \\
& & \qquad\qquad\qquad\qquad \leftarrow (\{b\}, \{a\}) \\
& a_6: & \mathit{acquire}\ \texttt{mutex} \\
& a_7: & \texttt{count\_a}-- \\
& a_8: & \texttt{count\_a} == 0 \\
& a_9: & \mathit{release}\ \texttt{lock} \\
& a_{10}: & \mathit{release}\ \texttt{mutex} \\
(\mathit{thread}\ b) \Rightarrow & b_1: & \mathit{acquire}\ \texttt{mutex} \\
& b_2: & \texttt{count\_b}++ \\
& b_3: & \texttt{count\_b} == 1 \\
& b_4: & \mathit{acquire}\ \texttt{lock} \\
& b_5: & \mathit{release}\ \texttt{mutex} \\
& b_6: & \mathit{acquire}\ \texttt{mutex} \\
& b_7: & \texttt{count\_b}-- \\
& b_8: & \texttt{count\_b} == 0 \\
& b_9: & \mathit{release}\ \texttt{lock} \\
& b_{10}: & \mathit{release}\ \texttt{mutex}
\end{array}
$$

Fig. 5. A possible interleaving of one class A thread and one class B thread

4 Our Methodology

4.1 Identifying Threads and Shared Objects Out of Multiple Runs

When `inspect` runs the program under test repeatedly, as the runtime environment may change across re-executions, each thread may not be allocated to the same id by the operating system. Also, dynamically-created shared objects may not reside in the same physical memory address in different runs.

We assumes that given the same external inputs, multiple threads in a program are always created in the same order. Banking on this, 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. In our implementation, we let each thread register itself in a mapping table, from system-allocated thread ids to integers. If the threads are created in the same sequential order in different runs, each thread will be assigned to the same id by this table. 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, the same shared objects between multiple runs can be recognized in a similar way as threads.

$$
\begin{array}{lll}
 & & \leftarrow (\{b\}, \{a\}) \\
(thread\ a) \Rightarrow & a_1 : acquire\ \texttt{mutex} & \\
 & a_2 : \texttt{count\_a} + + & \\
 & a_3 : \texttt{count\_a} == 1 & \\
 & a_4 : acquire\ \texttt{lock} & \\
 & a_5 : release\ \texttt{mutex} & \\
(thread\ b) \Rightarrow & b_1 : acquire\ \texttt{mutex} & \\
 & b_2 : \texttt{count\_b} + + & \\
 & b_3 : \texttt{count\_b} == 1 &
\end{array}
$$

Fig. 6. An interleaving results in deadlock.

4.2 Communicating With the Scheduler

As explained before, `inspect` works by having the instrumented threads calling the scheduler before executing visible operations, and moving forward only based on the permissions being granted by the scheduler. The requests that a thread can send to the scheduler can be classified into four classes: 1) thread-management related events, including thread creation, thread destruction, thread join and so on; 2) mutex-events, include mutex init, destroy, acquire and release; 3) read-write lock init, destroy, reader lock, writer lock, and unlock operations; 4) condition variable related events, including condition variable creation, destroy, wait, signal and broadcast; 5) data object related events, including creation of the data objects, and read and write operations.

4.3 Handling wait And signal

Condition variable and the related wait/signal/broadcast routines are a necessity in many threading libraries. The wait/signal routines usually obey the following rules: (i) The call to a condition wait routine shall block on a condition variable; (ii) Each shall be called with a mutex locked by the calling thread; (iii) The condition wait function atomically releases the mutex and causes the calling thread to block on the condition variable; and (iv) Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

A common problem (user bug) related to *wait* and *signal* operations is the "lost wake-up" caused by a thread executing a signal operation before the different waiter goes into the waiting status. This can cause the waiter to block forever, since condition signals do not pend. We now explain how `inspect` takes care of this *so as to not mask such bugs*.

`Inspect` handles the condition variable related events by splitting the wait routine into three sub-operations: `pre_wait`, `wait`, and `post_wait`. Here, `pre_wait` releases the mutex that is held by the caller thread; `wait` changes the thread into blocking status, and puts the thread into the waiting queue of the correspondent condition variable; and `post_wait` tries to re-acquire the mutex again. The wrapper function for the `pthread_cond_wait` routine is shown in Figure 7.

```
1: inspect_cond_wait(cond, mutex) {
2:     send pre_wait request;
3:     receive pre_wait permit;
4:     receive unblocking permit;
5:     send post_wait request;
6:     receive post_wait permit;
7: }
```

Fig. 7. Wapper function for `pthread_cond_wait`

As shown in Figure 7, when the wait routine is invoked, the calling thread t first sends a `pre_wait` request to the scheduler; the scheduler records that t releases the mutex, and sets t's status as blocking. The scheduler will not send an "*unblocking*" permit to t until some other threads send out a related signal, and t is picked out by the scheduler from the waiting queue. In t, after receiving the *unblocking* permit from the scheduler, t will send a `post_wait` request to acquire the mutex.

4.4 Avoiding Redundant Backtracking

In runtime model checking, backtracking is an expensive operation as we need to restart the program, and replay the program from the initial state until the backtrack point. To improve efficiency, we want to avoid backtracking as much as possible. Line 21 in Figure 2 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. Due to space limitation, the details are omitted here. More details are to be found in our technical report [16].

4.5 Automated Instrumentation

`Inspect` needs to capture every visible operation to guarantee that it is not missing any bug in the program. Incorrect instrumentation can make the scheduler fail to observe visible operations (viz., before executing some visible operations, the program under test does not notify the scheduler). To automate the instrumentation process, we designed an algorithm as shown in Figure 8.

The automated instrumentation is primarily composed of three steps: (i) replace the call to the thread library routines with the call to the wrapper functions; (ii) before each visible operation on a data object, insert code to send a request to the scheduler; (iii) add *thread start* at the entry of every thread, and *thread end* at each exit point.

To achieve this, we need to know whether an update to a data object is a visible operation or not. We use the may-escape analysis [17] to discover the shared variables among threads. Because the result of 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.

```
1:  auto_instrument(program P) {
2:      have an inter-procedural escape analysis on P to find out all the possible shared
            variables among threads;
3:      for each call of the thread library routines
4:          replace the call with the call to the correspondent wrapper function;
5:      for each  access of a shared variable v {
6:          if (read access)
7:              insert a read request for v before reading v;
8:          else
9:              insert a write request for v before updating v;
10:     }
11:     for each entry of threads
12:         insert a thread start notification to the scheduler, before the first statement
            of the thread;
13:     for each exit point of threads
14:         insert a thread end notification after the last statement of the thread;
15: }
```

Fig. 8. Automated instrumentation

4.6 Detecting Bugs

`Inspect` detects data races and deadlocks while updating the backtracking information. If two transitions on a shared data object are enabled in the same state, and at least one of them is a write operation, `inspect` will report a data race. Deadlocks are detected by checking whether there is a cycle in resource dependency. `Inspect` keeps a resource dependent graph among threads, checks and updates the graph before every blocking transition.

Besides races and deadlocks, `inspect` can also report incorrect usages of synchronization primitives. The incorrect usages include: (1) using an uninitialized mutex/condition variable; (2) not destroying mutex/condition variables after all threads exit; (3) releasing a lock that is held by another thread; (4) waiting on

the same condition variable with different mutexes; (5) missing a pthread-exit call at the end of function `main`.

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 more reliable. Messages will neither get lost nor delivered out of order.

In the automated instrumentation part, we first use CIL [18] as a preprocessor to simplify the code. Then we use our own program analysis and transformation framework based on `gcc`'s C front end to do the instrumentation. We first have an inter-procedural flow-sensitive alias analysis to compute the alias information. With the alias information, we use an inter-procedural escape analysis to discover the shared variables among threads. Finally we follow the algorithm in Figure 8 to do the source code transformation. Right now the automatic instrumentation can only work for C programs because of the lack of a front end for C++.

6 Experiments and Evaluation

We evaluate `inspect` on two sets of benchmarks. The first set includes two benchmarks in [1]. The second set contains several small applications that use pthread on `sourceforge.net` and `freshmeat.net`[19,20,21,22].

The performance of `inspect` for the benchmarks in [1] is shown in Table 1. The first program, *indexer*, captures the scenarios in which multiple threads insert messages into a hash table concurrently. The second benchmark, *fsbench*, is an abstraction of the synchronization idiom in Frangipani file system. We rewrote the code using C and the POSIX thread library. In the original *indexer* benchmark, a compare-and-swap is used. As C does not have such an atomic routine, we replaced it with a function and used a mutex to guarantee the mutual exclusion. The execution time was measured on a PC with two Intel Pentium CPUs of 3.0GHz, and 2GB of memory. `Inspect` was compiled with gcc-3.3.5 at optimization level `-O2`.

In Table 1, it shows that when the number of threads increases, more conflicts among threads slow down the program. However, `inspect` can still explore more than 30 different interleavings per second.

Table 1. Checking indexer and fsbench

	threads	runs	transitions	time(s)	runs/sec
indexer	1-11	1	$\leqslant 272$	0.01	-
	13	64	6,033	1.44	44.44
	14	512	42,635	12.58	40.69
	15	4,096	351,520	108.74	37.68
	16	32,768	2,925,657	988.49	33.15
fsbench	1-13	1	$\leqslant 209$	0.01	-
	16	8	1,242	0.14	-
	18	32	4,893	0.64	50
	20	128	20,599	2.76	46.38
	22	512	84,829	11.94	42.88
	24	2,048	367,786	54.82	37.36
	26	8,192	1,579,803	261.40	31.33

We also tried `inspect` on several real applications that use pthread on `sourceforge.net` and `freshmeat.net`. Table 2 shows the result. The application *aget*[19] is an ftp client in which multiple threads are used to download different segments of a large file concurrently. Application *pfscan*[20] is a multi-threaded file scanner that combines the functionality of `find`, `xargs`, and `fgrep`. It uses multiple threads to search in parallel through directories. Application *tplay*[21] is a multimedia player that uses one thread to prefetch the audio data, and the other thread to play the audio. Finally, *libcprops*[22] is a C prototyping tools library which provides thread-safe data structures such as linked list, AVL tree, hash list, as well as a thread pool and thread management framework.

In Table 2, the second column LOC (lines of code) for each application is counted with `wc`. The right most column shows the number of errors we found. We limited the inputs to have `inspect` explore a relatively small state space. For example, we tested `aget` on downloading a file of 3k bytes. This limits the total number of runs that `inspect` needs to explore to less than 6,000, and `inspect` can finish checking within 5 minutes. As `inspect` may report the same error multiple times while backtracking and re-executing the program repeatedly, we only count the unique number of errors.

For *aget*, we found one data race which is also reported in [6]. When testing *aget* with `inspect`, we need to construct a closed environment for it. As the network may introduce non-determinism to the environment, we reduced the size of the data package, which *aget* gets from the ftp server, to 512 bytes.

In *pfscan*, we found four errors. One error is that a condition variable is used without initialization. This is a dangerous behavior, and it may completely 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 results in resource leakage. Also, we found that a `pthread_exit` was missing at the end of main. As a result, when the main thread exits, some worker threads may be killed before they completely finish their work. .

Table 2. Checking real applications

benchmark		LOC	threads	Errors		
				races	deadlock	other errors
aget-0.4		1,098	3	1	0	0
pfscan-1.0		1,073	4	1	0	4
tplay-0.6.1		3,074	2	0	0	0
libcprops-0.1.6	avl	1,432	1-3	2	0	0
	heap	716	1-3	0	0	0
	hashlist	1,953	1-3	1	0	1
	linked_list	1,476	1-3	1	0	0
	splay_tree	1,211	1-3	1	0	0

As for *libcprops*, it is a thread-safe library and test drivers are required for testing it. We adapted the test cases in the *libcprops* release into multithreaded versions, and used them as test drivers. `Inspect` revealed several data races in the code. After manually examining the source code, we found that most of the races are benign races. Besides, we also found that in *hashlist*, a condition variable is destroyed without initialization. This may lead to undefined behaviors.

6.1 Discussion

Our experiments show that `inspect` can be very helpful in testing and debugging multithreaded C/C++ applications. However, it also has limitations. First, `inspect` needs a set of test cases incorporated in its test driver to get good coverage of the code being verified. Secondly, runtime monitoring puts an overhead on the program, especially in programs that have a lot of visible operations on shared data objects. Also, the intrusive instrumentation limits `inspect` from checking programs that have strict timing requirements. As `inspect` checks the program's behavior by monitoring the concrete executions of the program, it is not able to check system-level code as what RacerX, LockSmith, etc. can do.

It is clear that for `inspect` to check a program, we must be able to concretely execute the program. When doing our experiments, we also tried running several other open-source applications. Unfortunately, some problems were encountered: (i) some programs kept crashing because of other existing bugs; (ii) it is inconvenient to construct a closed world for server programs such as http servers. Other than these limitations, we think `inspect` is a powerful assistant tool in the process of unit testing and debugging for multithreaded software.

7 Other Related Work

Lei et al.[23] 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. Instead, dynamic partial order reduction views it as an interleaving of visible operations

from multiple threads. Compared with RichTest, `inspect` focuses on checking multithreaded C/C++ programs, and it can detect not only data races, but also deadlocks and other errors. However, `inspect` cannot yet handle send/receive events between multiple processes.

CMC[24] 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[25] 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.

jCute[26] uses a combination of symbolic and concrete execution to check a multithreaded Java program by feeding it with different inputs and replaying the program with different schedules. jCute is more powerful in discovering inputs that can have the program execution take different paths. We think the difference between our work and jCute is in the implementation part. jCute uses the Java virtual machine to intercept visible operations of a multithreaded Java program. Here we use socket communication and an external scheduler for C/C++ programs.

Helmstetter et al.[27] show how to generate scheduling based on dynamic partial order reduction. We think that the differences between our work and theirs lie in: (i) We are focusing on application-level multithreaded C programs, while they focused on the schedulings of SystemC simulations; and (ii) Instead of generating the scheduling only, our work reruns the program and tries to verify safety properties.

CHESS[28] is the work which is probably most similar to ours. As CHESS is not publicly available, we are only able to evaluate CHESS according to the available documentation. From our understanding, the main difference between CHESS and `inspect` lies in the instrumentation part and how to take control of the scheduling away from the operation system. In CHESS, the instrumentation allocates a semaphore for each thread that is created. It also requires an invariant to be preserved: at any time every thread but one is blocked on its semaphore. In contrast, `inspect` does source to source transformation, and uses blocking sockets to communicate between scheduler and the threads.

8 Conclusion

In this paper, we propose a new approach to model check safety properties including deadlocks and stuttering invariants in multithreaded C/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[29] 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.

References

1. 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.
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. 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.
8. 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.
9. 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.
10. 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.
11. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

12. Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *ASE*, pages 3–12, 2000.
13. Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.
14. David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1998.
15. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
16. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. UUCS-07-008:Runtime Model Checking of Multithreaded C/C++ Programs. Technical report, 2007. http://www.cs.utah.edu/research/techreports/2007/ps/UUCS-07-008.ps.
17. 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.
18. http://manju.cs.berkeley.edu/cil/.
19. http://freshmeat.net/projects/aget/.
20. http://freshmeat.net/projects/pfscan.
21. http://tplay.sourceforge.net/.
22. http://cprops.sourceforge.net/.
23. Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006.
24. 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.
25. 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.
26. Koushik Sen and Gul Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.
27. 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.
28. http://research.microsoft.com/projects/CHESS/.
29. 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.