RV-Predict[C] Documentation

Release 0.1

Runtime Verification Inc.

March 07, 2016

CONTENTS

1 Contents **3**
1.1 Quickstart 3
1.2 Running Examples 3

RV-Predict[C] is the only dynamic data race detector that is both sound and maximal. *Dynamic* means that it executes the program in order to extract an execution trace to analyze. *Sound* means that it only reports races which are real (i.e., no false positives). And *maximal* means that it finds all the races that can be found by any other sound race detector analyzing the same execution trace. The technology underlying RV-Predict is best explained in the following papers:

> Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14). ACM, New York, NY, USA, 337-348.
>
> Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. 2012. Maximal Causal Models for Sequentially Consistent Systems. In Proceedings of the 3rd International Conference on Runtime Verification (RV '12). LNCS Volume 7687, 2013, pp. 136-150

CHAPTER

ONE

CONTENTS

1.1 Quickstart

RV-Predict[C] works in two steps. First, `$ rv-predict-compile file.c` creates an instrumented version of a multithreaded program (rv-predict-compile is just a wrapper for our customized version of clang compiler). Second, `$ rv-predict-execute ./a.out` performs and offline data race analysis.

```
rv-predict-compile file.c
rv-predict-execute ./a.out
```

You can also use RV-Predict[C] with a piece of software built using Gnu Autoconf, use the following command (our tool currently relies on clang compiler for the generation of the instrumented code):

```
CC=clang CFLAGS=-fsanitize=rvpredict ./configure
```

You can also configure a makefile which has specified a CC variable for specifying the compiler with

```
make <target> CC=clang CFLAGS=-fsanitize=rv-predict
```

Note: if your code uses `g++` just replace `clang` with `clang++`.

1.2 Running Examples

We provide examples demonstrating RV-Predict capabilities in detecting concurrency bugs. Below we focus on detecting data races. Data races are a common kind of concurrency bug in multi-threaded applications. Intuitively, a data race occurs when two threads concurrently access a shared memory and at least one of the accesses is a write. Data races are very hard to detect with traditional testing techniques. It requires occurrence of simultaneous access from multiple treads to a particular region which results with a corrupted data that violates a particular user provided assertion or test case. Therefore, traditional software engineering testing methodology is inadequate because all tests passing most of the time with rare fails with mysterious rare message might create a false sense of reliability.

Despite all the effort on solving this problem, it remains a challenge in practice to detect data races effectively and efficiently. RV-Predict aims to change this undesired situation. Below we are summarizing some of the most common data races in C and C++ and show how to detect them with RV-Predict. The examples described below can be found in RV-Predict[C] distribution `examples/demo` directory. For any file in that directory, simply run `rv-predict-compile [file].c[pp]` to compile it, followed by `rv-predict-execute ./a.out` to execute it.

1.2.1 1. Concurrent Access to a Shared Variable

This is the simplest form of a data race, and also the most frequent in practice. The problem description is straightforward: multiple threads are accessing a shared variable without any synchronization.

POSIX Threads

Consider the following snippet of the code from `dot-product.c` that uses POSIX Threads library for multi-threading.

```
void *dotprod(void *arg) {
  int i, start, end, len ;
  long offset;
  float mysum, *x, *y;
  offset = (long)arg;

  len = dotstr.veclen;
  start = offset*len;
  end   = start + len;
  x = dotstr.a;
  y = dotstr.b;

  mysum = 0;
  for (i = start; i < end ; i++) {
    mysum += (x[i] * y[i]);
  }

  dotstr.sum += mysum;
  printf("Thread %ld did %d to %d:  mysum=%f global sum=%f\n",offset,start,end,mysum,dotstr.sum);

  return NULL;
}
```

The function dotprod is activated when the thread is created. All input to this routine is obtained from a structure of type DOTDATA and all output from this function is written into this structure. The benefit of this approach is apparent for the multi-threaded program: when a thread is created we pass a single argument to the activated function - typically this argument is a thread number. All the other information required by the function is accessed from the globally accessible structure.

```
int main (int argc, char *argv[]) {
  //  << code ommitted for brevity >>
  /* Create threads to perform the dotproduct  */
  for(i = 0; i < NUMTHRDS; i++) {
    pthread_create(&callThd[i], NULL, dotprod, (void *)i);
  }
  // << code ommitted for brevity >>
}
```

The main program creates threads which do all the work and then print out result upon completion. Before creating the threads, the input data is created. Each thread works on a different set of data. The offset is specified by `i`. The size of the data for each thread is indicated by `VECLEN` (not shown above, please see the complete source). Since all threads update a shared structure, there is a race condition. The main thread needs to wait for all threads to complete, it waits for each one of the threads.

RV-Predict[C] works in two steps. (Make sure you are in the directory examples/demo.) First, `$ rv-predict-compile dot-product.c` creates an instrumented version of a multi-threaded program that

computes a dot products. Second, `$ rv-predict-execute ./a.out` performs an offline analysis. The results of the analysis:

```
Thread 0 did 0 to 10:  mysum=10.000000 global sum=10.000000
Thread 1 did 10 to 20:  mysum=10.000000 global sum=20.000000
Thread 2 did 20 to 30:  mysum=10.000000 global sum=30.000000
Sum =  30.000000
Data race on global 'dotstr' of size 24 at 0x0000014b47a0 (a.out + 0x0000014b47b0): {{{
    Concurrent write in thread T3 (locks held: {})
 ---->  at dotprod dot-product.c:62
    T3 is created by T1
        at main dot-product.c:107

    Concurrent write in thread T2 (locks held: {})
 ---->  at dotprod dot-product.c:62
    T2 is created by T1
        at main dot-product.c:107
}}}

Data race on global 'dotstr' of size 24 at 0x0000014b47a0 (a.out + 0x0000014b47b0): {{{
    Concurrent read in thread T2 (locks held: {})
 ---->  at dotprod dot-product.c:62
    T2 is created by T1
        at main dot-product.c:107

    Concurrent write in thread T3 (locks held: {})
 ---->  at dotprod dot-product.c:62
    T3 is created by T1
        at main dot-product.c:107
}}}

Data race on global 'dotstr' of size 24 at 0x0000014b47a0 (a.out + 0x0000014b47b0): {{{
    Concurrent write in thread T2 (locks held: {})
 ---->  at dotprod dot-product.c:62
    T2 is created by T1
        at main dot-product.c:107

    Concurrent read in thread T3 (locks held: {})
 ---->  at dotprod dot-product.c:63
    T3 is created by T1
        at main dot-product.c:107
}}}
```

First, note that the standard testing would not caught data races, because the output and the final result are as expected. However, RV-Predict's output correctly predicts three possible data races. The first one is on line 62: `dotstr.sum += mysum;`, where data race occurs because two threads can concurrently write to the shared variable. The second data race is concerned with the same line, however this time our analysis informs that data race exists due to a concurrent read and a concurrent write. Finally, the third report describes the case where there can be a concurrent write at line 62, and a concurrent read at line 63: `printf("Thread %ld did %d to %d:  mysum=%f global sum=%f\n",offset,start,end,mysum,dotstr.sum);`.

This example also showcases the maximality and predictive power of our approach. In particular, consider analysis results on the same program by widely used ThreadSanitizer tool from Google.

```
Thread 0 did 0 to 10:  mysum=10.000000 global sum=10.000000
==================
WARNING: ThreadSanitizer: data race (pid=6010)
  Write of size 4 at 0x0000014ae3b0 by thread T2:
    #0 dotprod /home/eddie/work/rv-predict-c/examples/demo/dot-product.c:62:14 (a.out+0x0000004a53cd)
```

```
  Previous write of size 4 at 0x0000014ae3b0 by thread T1:
    #0 dotprod /home/eddie/work/rv-predict-c/examples/demo/dot-product.c:62:14 (a.out+0x0000004a53cd)

  Location is global 'dotstr' of size 24 at 0x0000014ae3a0 (a.out+0x0000014ae3b0)

  Thread T2 (tid=6013, running) created by main thread at:
    #0 pthread_create /home/eddie/work/llvm-3.7.0.src/projects/compiler-rt/lib/tsan/rtl/tsan_intercep
    #1 main /home/eddie/work/rv-predict-c/examples/demo/dot-product.c:107:5 (a.out+0x0000004a5668)

  Thread T1 (tid=6012, finished) created by main thread at:
    #0 pthread_create /home/eddie/work/llvm-3.7.0.src/projects/compiler-rt/lib/tsan/rtl/tsan_intercep
    #1 main /home/eddie/work/rv-predict-c/examples/demo/dot-product.c:107:5 (a.out+0x0000004a5668)

SUMMARY: ThreadSanitizer: data race /home/eddie/work/rv-predict-c/examples/demo/dot-product.c:62:14 i
==================
Thread 1 did 10 to 20:  mysum=10.000000 global sum=20.000000
Thread 2 did 20 to 30:  mysum=10.000000 global sum=30.000000
Sum =  30.000000
ThreadSanitizer: reported 1 warnings
```

Note, that ThreadSanitizer can only detect one data race, specifically, the case when there are two concurrent writes to the shared variable.

Furthermore, consider Helgrind, another widely used tool for detecting concurrency bug that is part of the Valgrind tool-set. The result of Helgrind analysis is shown below.

```
Thread 0 did 0 to 10:  mysum=10.000000 global sum=10.000000
==6192== ---Thread-Announcement------------------------------------------
==6192==
==6192== Thread #3 was created
==6192==    at 0x515543E: clone (clone.S:74)
==6192==    by 0x4E44199: do_clone.constprop.3 (createthread.c:75)
==6192==    by 0x4E458BA: create_thread (createthread.c:245)
==6192==    by 0x4E458BA: pthread_create@@GLIBC_2.2.5 (pthread_create.c:611)
==6192==    by 0x4C30E0D: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==6192==    by 0x40090F: main (dot-product.c:107)
==6192==
==6192== ---Thread-Announcement------------------------------------------
==6192==
==6192== Thread #2 was created
==6192==    at 0x515543E: clone (clone.S:74)
==6192==    by 0x4E44199: do_clone.constprop.3 (createthread.c:75)
==6192==    by 0x4E458BA: create_thread (createthread.c:245)
==6192==    by 0x4E458BA: pthread_create@@GLIBC_2.2.5 (pthread_create.c:611)
==6192==    by 0x4C30E0D: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==6192==    by 0x40090F: main (dot-product.c:107)
==6192==
==6192== ----------------------------------------------------------------
==6192==
==6192== Possible data race during read of size 4 at 0x601080 by thread #3
==6192== Locks held: none
==6192==    at 0x4007E4: dotprod (dot-product.c:62)
==6192==    by 0x4C30FA6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==6192==    by 0x4E45181: start_thread (pthread_create.c:312)
==6192==    by 0x515547C: clone (clone.S:111)
==6192==
==6192== This conflicts with a previous write of size 4 by thread #2
==6192== Locks held: none
```

```
==6192==    at 0x4007F5: dotprod (dot-product.c:62)
==6192==    by 0x4C30FA6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==6192==    by 0x4E45181: start_thread (pthread_create.c:312)
==6192==    by 0x515547C: clone (clone.S:111)
==6192==  Address 0x601080 is 16 bytes inside data symbol "dotstr"
==6192==
==6192== ----------------------------------------------------------------
==6192==
==6192== Possible data race during write of size 4 at 0x601080 by thread #3
==6192== Locks held: none
==6192==    at 0x4007F5: dotprod (dot-product.c:62)
==6192==    by 0x4C30FA6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==6192==    by 0x4E45181: start_thread (pthread_create.c:312)
==6192==    by 0x515547C: clone (clone.S:111)
==6192==
==6192== This conflicts with a previous write of size 4 by thread #2
==6192== Locks held: none
==6192==    at 0x4007F5: dotprod (dot-product.c:62)
==6192==    by 0x4C30FA6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==6192==    by 0x4E45181: start_thread (pthread_create.c:312)
==6192==    by 0x515547C: clone (clone.S:111)
==6192==  Address 0x601080 is 16 bytes inside data symbol "dotstr"
==6192==
Thread 1 did 10 to 20:  mysum=10.000000 global sum=20.000000
Thread 2 did 20 to 30:  mysum=10.000000 global sum=30.000000
Sum =  30.000000
```

Helgrind is able to detect two data races related to concurrent writes or a concurrent read and a concurrent write at line 62, but not is not able to predict with a concurrent write at line 62 and a concurrent read at line 63.

C/C++ 11

One of the most significant features in the new C and C++11 Standard is the support for multi-threaded programs. This the feature makes it possible to write multi-threaded C/C++ program without relying on platform specific extensions and writing portable multi-threaded code with standardized behavior. RV-Predict[C] support C/C++11 concurrency, and thus it is able to detect concurrency bugs in the code written using C/C++11 constructs.

Consider the following example implementing a simple state machine.

```
mutex l;
bool ready = false;
enum State { STOP, INIT, START };
State state = STOP;

void init() {
  l.lock();
    ready = true;
  l.unlock();
  state = INIT;
  l.lock();
    ready = true;
  l.unlock();
}

void start() {
  // yield increases likelihood of avoiding expensive locking and unlocking
  // before being ready to enter the START state
```

```
  this_thread::yield();
  l.lock();
    if (ready && state == INIT) {
      state = START;
   }
  l.unlock();
}

void stop() {
  l.lock();
    ready = false;
    state = STOP;
  l.unlock();
}

int main() {
    thread t1(init);
    thread t2(start);
    thread t3(stop);
    t1.join(); t2.join(); t3.join();
    return 0;
}
```

(For full source see examples/demo/simple-state-machine.cpp.) This program implements state machine with three states, and each thread models some state machine transitions. Moreover, the developers seem to have devised a reasonable locking policy that appears to protect shared resources. This class of programs are hard to test, since there are many valid observable behaviors. So, some of the previously mentioned tools ThreadSanitizer or Helgrind can be used to increase confidence in the correctness of the program. In fact, neither ThreadSanitizer nor Helgrind report any problems with programs.

However, there are three subtle data races in the program, and RV-Predict[C] finds them all.

```
Data race on global 'state' of size 4 at 0x00000153ccf4 (a.out + 0x00000153ccf4): {{{
    Concurrent write in thread T2 (locks held: {})
 ---->  at init() simple-state-machine.cpp:19
    T2 is created by T1
        at main simple-state-machine.cpp:44

    Concurrent read in thread T3 (locks held: {WriteLock@94})
 ---->  at start() simple-state-machine.cpp:28
        - locked WriteLock@94 at start() simple-state-machine.cpp:27
    T3 is created by T1
        at main simple-state-machine.cpp:44
}}}
```

First data race is due to a write at line 19: `state = INIT;`, while concurrently reading the current value of the state variable. This behavior might lead to a behavior where the START state is not reached because of the aforementioned data race.

```
Data race on global 'state' of size 4 at 0x00000153ccf4 (a.out + 0x00000153ccf4): {{{
    Concurrent write in thread T2 (locks held: {})
 ---->  at init() simple-state-machine.cpp:19
    T2 is created by T1
        at main simple-state-machine.cpp:44

    Concurrent write in thread T4 (locks held: {WriteLock@94})
 ---->  at stop() simple-state-machine.cpp:37
        - locked WriteLock@94 at stop() simple-state-machine.cpp:35
    T4 is created by T1
```

```
        at main simple-state-machine.cpp:45
}}}
```

Second data race is likely particularly dangerous, because there are concurrent writes of INIT and STOP to the state variable, which effectively means that the program could begin entering the START state with possibly critical reasons to prevent the progress.

```
Data race on global 'state' of size 4 at 0x00000153ccf4 (a.out + 0x00000153ccf5): {{{
    Concurrent write in thread T2 (locks held: {})
 ---->  at init() simple-state-machine.cpp:19
    T2 is created by T1
        at main simple-state-machine.cpp:44

    Concurrent write in thread T3 (locks held: {WriteLock@94})
 ---->  at start() simple-state-machine.cpp:29
        - locked WriteLock@94 at start() simple-state-machine.cpp:27
    T3 is created by T1
        at main simple-state-machine.cpp:44
}}}
```

Finally, the third data race can effectively invert the state from START of INIT.

In summary, this simple program demonstrates that the state-of-the-art tools can be inadequate in detection of subtle data races with possibly dire consequences, while RV-Predict[C] can clearly identify all the data races.

1.2.2 2. Unsafe Data Strucuture Manipulation

Many standard library data structures are not designed to be used in a multi-threaded environment, e.g. widely used vector class.

First, consider a simple example (examples.demo/unsafe-vector.c):

```
#include <vector>
#include <thread>

using namespace std;

vector<int> v;

void thread1() {
    v.push_back(1);
}

void thread2() {
    v.push_back(2);
}

int main() {
    thread t1(thread1);
    thread t2(thread2);

    t1.join();
    t2.join();

    return 0;
}
```

In the example both threads are trying to add to `std::vector` without synchronization. RV-Predict[C] catches the data race as shown below.

```
Data race on global 'v' of size 24 at 0x00000153ecc8 (a.out + 0x00000153ecd8): {{{
    Concurrent read in thread T2 (locks held: {})
 ---->  at thread1() unsafe-vector.cpp:12
    T2 is created by T1
        at main unsafe-vector.cpp:20

    Concurrent write in thread T3 (locks held: {})
 ---->  at thread2() unsafe-vector.cpp:16
    T3 is created by T1
        at main unsafe-vector.cpp:20
}}}

...
```

This example is easily fixed by using some synchronization mechanisms (e.g., locks) when performing the access to the shared variable `v`.

Consider now a more interesting example (see below), where we used `vector` data structure to implement a stack. At first sight, it looks like all the operations are properly synchronized, however just because we are using a mutex or other synchronization mechanism to protect shared data, it does not mean we are protected from race conditions!

```
using namespace std;
mutex myMutex;
class stack
{
public:
  stack() {};
  ~stack() {};
  void pop();
  int top() { return data.back(); }
  void push(int);
  void print();
  int getSize() { return data.size(); }
private:
    vector<int> data;
};

void stack::pop()
{
  lock_guard<mutex> guard(myMutex);
  data.erase(data.end()-1);
}

void stack::push(int n) {
  lock_guard<mutex> guard(myMutex);
  data.push_back(n);
}

void stack::print()
{
  cout << "initial stack : " ;
  for(int item : data)
      cout << item << " ";
  cout << endl;
}
```

```
void process(int val, string s) {
  lock_guard<mutex> guard(myMutex);
  cout << s << " : " << val << endl;
}

void thread_function(stack& st, string s) {
  int val = st.top();
  st.pop();
  process(val, s);
}

int main()
{
    stack st;
    for (int i = 0; i < 10; i++)  st.push(i);

    st.print();

    while(true) {
      if(st.getSize() > 0) {
        thread t1(&thread_function, ref(st), string("thread1"));
        thread t2(&thread_function, ref(st), string("thread2"));
        t1.join();
        t2.join();
      } else break;
    }

    return 0;
}
```

(For full source see examples/demo/stack.cpp.) In the example below each shared access is guarded using

```
lock_guard<mutex> guard(myMutex);
```

Now, it would be tempting to conclude that the code is thread-safe. However, we actually cannot rely on the result of getSize(). Although it might be correct at the time of call, once it returns other threads are free to access the stack and might push() new elements to the stack or pop() existing elements of the stack.

This particular data race is consequence of the interface design, and the use of mutex internally to protect the stack does not prevent it. As shown below, RV-Predict[C] can be used to detect these kind of flaws.

```
Data race on array element #11: {{{
    Concurrent read in thread T3 (locks held: {})
 ---->  at stack::top() Stack.cpp:18
    T3 is created by T1
        at main Stack.cpp:66

    Concurrent write in thread T2 (locks held: {WriteLock@27})
 ---->  at stack::pop() Stack.cpp:29
        - locked WriteLock@27 at stack::pop() Stack.cpp:29
    T2 is created by T1
        at main Stack.cpp:65
}}}
```

1.2.3 3. Double-checked Locking

Suppose you have a shared resource (e.g.shared a database connection or a large allocation a big chunk of of memory) that is expensive to construct, so it is only done when necessary. A common idiom used in such cases is known as

double-checked locking pattern. The basic idea is that the pointer is first read without acquiring the lock, and the lock is acquired only if the pointer is NULL. The pointer is then checked again once the lock has been acquired in case another threads has done the initialization between the first check and this thread acquiring a lock.

For full source see examples/demo/double-checked-locking.cpp.

```
struct some_resource
{
    void do_something()
    {}

};

std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;
std::thread thread;
std::thread join;
void foo()
{
  if(!resource_ptr) {
    std::unique_lock<std::mutex> lk(resource_mutex);
    if(!resource_ptr)
    {
        resource_ptr.reset(new some_resource);
    }
    resource_ptr->do_something();
  }
}

int main()
{
    std::thread::thread t1(foo);
    std::thread::thread t2(foo);

    t1.join();
    t2.join();
}
```

However, this pattern has become infamous because it has potential for a nasty race condition. As shown below, RV-Predict[C] detect the race condition. Specifically, the data race occurs because the read outside the lock is not synchronized with the write done by the thread inside the lock. The race condition includes the pointer and the object pointed to: even if a thread sees the pointer written by another thread, it might not see the newly created instance of `some_resource`, resulting in the call to `do_something()` operating on incorrect values.

```
Data race on global 'resource_ptr' of size 16 at 0x00000153dcc8 (a.out + 0x00000153dcc8): {{{
    Concurrent read in thread T3 (locks held: {})
 ---->  at foo() double-checked-locking.cpp:19
    T3 is created by T1
        at main double-checked-locking.cpp:32

    Concurrent write in thread T2 (locks held: {WriteLock@dc})
 ---->  at foo() double-checked-locking.cpp:23
        - locked WriteLock@dc at foo() double-checked-locking.cpp:21
    T2 is created by T1
        at main double-checked-locking.cpp:32
}}}
...
```

1.2.4 4. Broken Spinnning Loop

Sometimes we want to synchronize multiple threads based on whether some condition has been met. And it is a common pattern to use a while loop that repeatedly checks that condition:

```
using namespace std;

bool condition = false;
int sharedVar;

void thread1() {
    sharedVar = 1;
    condition = true;
}

void thread2() {
    while(!condition) {
        this_thread::yield();
    }
    if(sharedVar != 1) {
        throw new runtime_error("How is this possible!?");
    }
}

int main() {
    thread t1(thread1);
    thread t2(thread2);
    t1.join();
    t2.join();
    return 0;
}
```

As shown below, RV-Predict[C] detect the data race on `condition` variable.

```
Data race on global 'condition' of size 1 at 0x00000153cd88 (a.out + 0x00000153cd88): {{{
    Concurrent write in thread T2 (locks held: {})
 ---->  at thread1() spinning-loop.cpp:14
    T2 is created by T1
        at main spinning-loop.cpp:28

    Concurrent read in thread T3 (locks held: {})
 ---->  at thread2() spinning-loop.cpp:18
    T3 is created by T1
        at main spinning-loop.cpp:28
}}}
```