Verification of parallel systems

August 2016

Francis Giraldeau
francis.giraldeau@polymtl.ca

Prof. Michel Dagenais
michel.dagenais@polymtl.ca

Polytechnique Montréal

Verification methods

- Testing
  - Easy, but a test is not a proof
  - If a race condition happens rarely, a test may not find it
- Runtime verification
  - Instrument the program and run it
  - Direct verification of a specific implementation
  - Must execute the code to verify it
- Static verification
  - Types and value bounds
  - State machine model
  - State space exploration to verify properties (i.e. deadlock is not possible)
  - Difficult to verify large software in practice: state space is exponential
  - The model may be valide, but an implementation error can still exist

Basic tests for parallel systems

- Run the serial algorithm
- Run the parallel algorithm
- Assert that both yield the same result
- Vary the number of threads
- Check for odd computation bounds
  - Empty interval
  - Interval smaller than number of threads
  - Check for overflow eith large intervals
- Measure the speedup

Memory accesses verification

- Mainly using shadow memory
- Intercep all read (load) and write (store) of a program (mov to and from memory on Intel)
- Intercept calls to malloc()/free()
- Intercept locking events
- Maintain state about each byte
  - Allocated: read or write invalid
  - Initialized: read uninitialized memory
  - Concurrency: concurrent access by at least two threads where one is a write. Intersection of lock-set for a memory location must not be empty (there exists a lock preventing concurrent access to the memory)

Race detection

- Thread 1
  - lock (&lock1)
  - mov (0x123),%rax
  - add $1, %rax
  - mov %rax,(0x123)
  - unlock (&lock1)
- Thread 2
  - lock (&lock2)
  - mov (0x123),%rax
  - add $1, %rax
  - mov %rax,(0x123)
  - unlock (&lock2)

Lock set thread 1 (LS1) = 0x123 : { lock1 }
Lock set thread 2 (LS2) = 0x123 : { lock2 }

$$LS_1 \cap LS_2 = \emptyset$$

The set is empty, meaning no common lock protects agains a race condition for that memory address

Runtime deadlock detection

- Record the order of locks
- Detect cycles in the resulting graph
- Implementations:
  - Linux kernel lockdep
  - Valgrind helgrind
  - Clang ThreadSanitizer

A word about floating point

- Floating point aithmetic is not associative

```
void FpTest::testFloatingPointAssociative()
{
    float x = -1.5 * pow(10, 38);
    float y = 1.5 * pow(10, 38);
    float z = 1.0;

    float r1 = x + (y + z);
    float r2 = (x + y) + z;

    qDebug() << "r1" << r1 << "r2" << r2;

    QVERIFY2(r1 == r2, "result is not associative");
}

/* sortie:
   r1 0 r2 1
   FAIL!  : FpTest::testFloatingPointAssociative()
   'r1 == r2' returned FALSE. (result is not associative)
*/
```

$$x + (y + z) \neq (x + y) + z$$

Compile-time check

- Inspection of syntax tree
- Propagate variable values
- Set of rules
  - Division by zero
  - Null pointer
  - Out-of-bound access
- Implementation: clang scan-build

http://clang-analyzer.llvm.org/scan-build.html

Formal verification

- Make a state machine model of the software
- Explore the state space to verify a property
- Temporal logic
  - E<> p : There exist a path that satisfies property p
  - E[] p : There exists a path that satisfies eventually always p
  - A<> p : All path satisfies p eventually
  - A[] p : Property p is always satisfied
  - p --> q : if property p is satisfied, the eventually property q will be satisfied
- Implementation: UPPAAL
  - Race condition
  - Deadlock
  - Worst Case Execution Time

Examples

- 40-race: race condition example
- 41-memcheck: invalid memory accesses
- 42-deadlock: lock verification
- 43-bitfield: race condition on bitfield
  - valgrind –tool=helgrind
  - valgrind --tool=memcheck
  - clang++ -sanitize=thread
  - clang++ -sanitize=address
  - scan-build make