CS636 Course Project
WCP Analysis

Antriksh Gupta (210159)
Kumar Harsh Mohan (210543)
Shourya Trikha (210994)

April 21, 2025

Overview

This project implements a dynamic analysis tool using the Intel PIN dynamic binary instrumentation framework to detect predictable data races in multithreaded programs based on the Weak Causal Precedence (WCP) model.

The WCP model is a partial-order relation that extends the traditional happens-before (HB) relation by relaxing certain ordering constraints, allowing it to detect a wider class of data races while avoiding false positives. Compared to HB, WCP provides greater coverage of potential concurrency bugs without sacrificing soundness.

Compilation and Running

We name the PIN root directory (containing the `pin` executable) as `DIR` (It could be something like `~/pin-external-3.31-98869-gfa6f126a8-gcc-linux`).
Create a new directory `DIR/source/tools/WCPPinTool` and place the following in the newly created directory:

- `WCPPinTool.cpp`
- `makefile, makefile.rules`
- `run.sh, test.sh`
- `include` directory
- `test` directory

Replace the value of PIN root directory on line 4 of `run.sh` by `DIR`. Create a directory `obj-intel64` in the `WCPPinTool` directory and use `make obj-intel64/WCPPinTool.so` to compile the PIN tool as follows:

```
DIR/source/tools/WCPPinTool$ mkdir obj-intel64
DIR/source/tools/WCPPinTool$ make obj-intel64/WCPPinTool.so
```

Use the following command to run the WCP Analysis for an executable. The maximum number of threads used by the target program should be provided using the `-t` flag (the default is 8). An output file can be optionally provided (the default is `wcp_analysis.out`). The output file contains information about detect data races (if any):

```
DIR/source/tools/WCPPinTool$ ./run.sh <executable> [-t <maxThreads>] [-o <output>]
```

To run all the tests, Use the `test.sh` script. It creates an `output` directory and places the results of the analysis in it.

```
DIR/source/tools/WCPPinTool$ ./test.sh
```

Note: The macro `LOG_TRACE` can be defined in the `WCPPinTool.cpp` file to generate a trace of the processed events in the file `trace.out`

Implementation Details

The analysis is performed on a 4-byte granularity, meaning that each 4-byte aligned memory location is considered a different variable. We assume that Reads and Writes happen on these 4-byte memory locations.

Tracking events

- Read and Write operands of an instruction are used to track memory events. The events are processed just before the actual read/write.
- Lock acquire and release events are tracked by instrumenting various acquire and release routines from the `pthread` library. More specifically, the following routines are instrumented:
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `pthread_spin_lock`
  - `pthread_spin_unlock`
  - `pthread_rwlock_wrlock`
  - `pthread_rwlock_unlock`
  - `pthread_mutex_trylock`
  - `pthread_spin_trylock`
  - `pthread_rwlock_trywrlock`

  The acquire event is processed after the acquire routine returns and the release event is processed just before the actual call to the release routine. In case of try locks, the return value is checked to determine whether lock acquire was successful. In case of `pthread_rwlock`, the acquires and releases of the read-lock are not tracked since they don't create mutual exclusion.
- Thread fork is tracked by instrumenting the `pthread_create` routine. After spawning the thread, the parent waits till the child has inherited information from the parent's vector-clocks. This parent-child synchronization during thread fork is done using atomic flags.
- Thread join is not tracked explicitly as in case of thread fork. This is due to the fact that, for some reason, PIN was unable to instrument `pthread_join`. Hence, we consider the exit of a child thread as the join event. This is handled in the `ThreadFini` function which executes once a thread terminates. The parent inherits the vector-clock information from the exited child threads and this synchronization is again done using atomic flags.

Notes:

1. Since we are not tracking thread join explicitly, it can sometimes miss data races or detect spurious data races which would not be the case if thread join could be tracked explicitly. It should however be noted that the detected races don't contain any false positives or false negatives given the event trace that the tool sees. It is just that the parent thread can see the join event earlier or later than the actual thread join (this is because the child thread can terminate well before the parent actually joins the child; on the other hand it may happen that the parent joins the child however the `ThreadFini` function for the child gets delayed).

2. It is assumed that lock acquires and releases are well nested, i.e. if lock B is acquired after lock A, then lock B must be released before the release of lock A. If this condition is violated, the addresses of the violating locks is printed and the analysis terminates.

Shadow Memory

To store the metadata (read/write vector clock for variables, Acq/Rel queues, vector clocks and other metadata for locks), a shadow memory location for each 4 byte memory location is maintained. The shadow memory structure is a concurrent hash table with 2^{42} buckets. Since for a 48-bit virtual address system, the maximum size of user space address space is 2^{47} bytes and we are working on a 4-byte granularity, each bucket can contain a maximum of 8 elements. In practice, the pointers to the buckets containing metadata are stored in the shadow memory array. It should be noted that the shadow memory is allocated using `mmap` with the flags `MAP_ANONYMOUS | MAP_NORESERVE`. As a result, we get lazy allocation of memory which is automatically zero-initialized and the kernel doesn't need to reserve swap space corresponding to the huge amount of virtual memory allocated. The memory footprint of the shadow memory thus remains proportional to the memory usage of the target program.

We use the least significant bit of the bucket pointers as a mark bit that tells whether the bucket is currently being accessed by a thread along with atomic primitives provided by GCC (`__atomic_compare_exchange_n`, `__atomic_load_n` and `__atomic_store_n`) to ensure mutual exclusion while accessing the buckets.