Project Overview

Our project is a state space equivalence check for multithreading scenarios. Given any arbitrary functions, our project provides a wrapper that determines if the functions are thread-safe and can be interleaved in any arbitrary way while still producing an identical memory state at the end as when the functions are run in a sequential manner.

Memory State Checking

We check the memory state to verify that the outcome of an interleaved schedule obeys the outcome of a sequential schedule. The user provides the memory address and nbytes that they want to verify the consistency of the state of. This is placed into a memory state wrapper that we implemented.

Our wrapper provides some simple functions that allow us to easily initialize, capture, reset, and verify the memory state.

Our code will run all possible sequential schedules (n! schedules, where n = number of functions/threads), and hash + save the memory state at the end of each of the schedules. These are saved as our acceptable valid end states.

After this, we run all of our interleaved schedules, and verify the memory state of each one by comparing the hash at the end to the valid hashes we generated initially. A hash mismatch indicates that we have found an invalid end state with the current interleaved schedule, and we highlight this for the user.

Schedule Generation

Our scheduler carefully considers all possible interleavings of instructions across functions, ensuring that sequential consistency is only maintained within the context of a single function - in other words, the interleavings our scheduler generates are guaranteed to not reorder instructions within a function, but run instructions from different functions in any order (indeed all possibilities). Our scheduler takes in parameters like number of context switches and number of functions from which to interleave instructions.

After generating the interleaved instruction schedules, we have to ensure that we are able to properly follow each interleaving in our context switching infrastructure - in order to do this, we convert (and indeed condense, saving a significant amount of memory) each interleaving into an array of length equal to the number of desired context switches - specifically, this array specifies what instruction each function should context switch on. As noted in other sections of this report, our code does indeed break at every instruction, in order to allow for the possibility of maximum,

worst-case interleaving - however, we carefully add conditionals to the context switch to ensure that real context switching simulated follows the scheduling given by any one interleaving.

Thread Schedule Wrapper

We built a wrapper around a simplistic version of threads, which allows us to run any arbitrary user-provided (or programmatically generated) schedule with them (from fully sequential schedules to fully interleaved ones, and anything in between). We maintain a queue of all running threads, and run each thread until a specified context switch: at this point we halt the current thread and switch to the next thread specified in the schedule.

Our first implementation of the wrapper had two main issues:

1) It would launch new threads for each new schedule (so if the user provides 10 unique schedules, we would launch 10 threads). This resulted in large memory use due to each thread having its own stack (and the lack of a kfree implementation), and when testing many possible interleavings we would run out of memory.
2) We would switch on any instruction, which included things like adds or multiplies. If a function only did one load and one store, but had 98 arithmetic instructions, our scheduler would do interleavings with all 100 total instructions. This led to massive combinatorial explosion, and runtimes for our checker would become extremely large due to the high volume of interleavings that it would have to check.

We solved these issues as follows:

1) We updated our wrapper to only launch 1 thread per executable, and then reuse these threads for each new schedule run. This was done by resetting the threads states and registers after the completion of a schedule. After this, our memory footprint was minimal.
2) Using page A3-2 of the ARM manual, we decode instruction types and only switch on load/stores, since these operations are the only ones that can cause invalid states when interleaved. Continuing from our previous example, this approach results in the number of possible instructions we have to interleave decreasing from 100 to 2. This resulted in a massive speedup for our checker.

In general, we also had some pretty nasty bugs but subtle in our implementation that were hard to debug, which ended up being a nontrivial challenge. We struggled with a bug where we were getting an invalid cpsr state for a while, and chased it down to a minor error in the logic for how we were switching between threads in a user-provided schedule.

Locking and Atomics

When starting the project, we had hoped we'd be able to make use of std atomics from the C/C++ library to be able to put together robust, thread-safe and incorrect, thread-unsafe

programs and simulate running them on a multi-core machine with our context switching + instruction interleaving infrastructure. Unfortunately, we found out that this was not supported on the Raspberry Pi, due to absent hardware support. As a hack, we thought it would be cool to implement our own atomics in assembly. We implemented a spin-lock API in assembly (test lock, acquire lock, release lock) - given that the hardware did not give us atomic instructions for free, we needed to make sure our custom atomics were indeed safe, which we ensured by turning on/off interrupts before/after the lock API functions.

Miscellaneous Optimizations

In our initial approach, we would precompute all possible schedules and then run each one. When there were many functions and/or these functions had many instructions, generating the schedules would take a nontrivial amount of time. To speed this up, we also implemented an approach in which we generate new schedules on the fly and then run the schedule immediately: we then stop schedule generation upon the first detected invalid end state. This would get us results faster for many functions/long function use-cases.

We were also working on automatically generating a read/write set using a memory trap with the VM code, but unfortunately were not able to fully implement this by the deadline. We had marked our entire heap as being no_access, such that whenever a thread tried to access memory it would trigger a data fault handler. This data fault handler would then add the memory location to the read/write set, and then return to executing the thread. However, we had some bugs in our implementation of this that we weren't able to fix in time, mostly related to the data fault handler being triggered outside of the instructions within the threads which would mess up our states. This feature would have been nice though, as it would have removed the need for the user to specify the memory locations they care about manually.