TITLE

Fast Parallel Fuzzing of RISCV32 Programs
Jordi Gonzalez & Ajax Shung

URL

URL GOES HERE

SUMMARY

We are going to build a highly distributed fuzzer (capable of scaling to massive clusters) that is also highly parallel at the node level. We intend to exploit things like CUDA for GPU-offloading, vector intrinsics for CPU-workers, MPI for communication, etc., to extract all available performance on a cluster, something the state of the art cannot yet do.

BACKGROUND AND CHALLENGES

We're going to be implementing a fuzzer for testing RISCV32 programs. A fuzzer is just a program that tests another target program by feeding it mutated or unexpected inputs and looking to see if the program executes new branches (or crashes, or fails to terminate in some reasonable time). The idea is to catch bugs (often security vulnerabilities) that are only revealed in corner cases.

Of course, to make sure a program is bug free we'd love to just simply test it on all possible inputs in the input space. For simple programs this is possible but for a program that, say, reads in a jpeg file and manipulates it, the space of input files is extremely large. So the goal of a fuzzer is instead to maximize the number of "new behaviors" (new code execution, crashes, etc.) found with the given compute resources. There are two axes for doing this:

- Increase the amount of inputs tested per unit time.
- Increase the amount of bugs that are found per input tested (ie choose inputs that are more likely to generate new behavior).

These align with our two axes of parallelism. First, we're going to make two types of emulators for RISCV32 programs. We chose RISCV32 because it's a small, simple ISA that we can actually emulate using our hardware:

- An emulator on the CPU that uses AVX512 to run an instance of the target program in each lane of the vector. This means each CPU thread will be able to run 16 instances of the target program, and then with multithreading on a multicore CPU we can parallelize many executions of the target program at once!
- An emulator on the GPU that uses CUDA to run many threads of the program at once, probably one on each CUDA thread.

In this way, we can utilize both GPUs and CPUs as workers to run fuzzing tests on our program. You might figure that, since the individual instances of the program don't have any dependencies on each other, this would be trivial to parallelize. This is not the case mostly because of branch divergence. So we want to group test cases that we think will take similar paths through the code together so that our SIMD instructions can actually do work for all the emulated threads at once (as they diverge we'd basically mask all but one lane and we'd basically be stuck executing one thread at a time). Of course, making this fast is also going to be hard. Remember, since we're emulating, we're free to take the assembly we're reading and change it so it's still correct but so that it's more performant on our emulating hardware.

The second axis of parallelism is quite tricky. Here's the general problem: let's say our strategy for generating new test inputs is to just start from a valid input and apply mutations. These mutations could be like flipping random bits or they could be like replacing four bytes with INT_MAX or stuff like that. This strategy is quite simple, but many of the tests it generates will be… completely useless! Let's say the target program expects a checksum or something, well, it's likely most of our tests will simply terminate with the checksum not being correct. The general problem here is that the program might have a very reliable parser (that has already been tested) and the bugs we're looking to find are only going to be exposed when we have a "valid" input that makes it through the parser and into the main program logic.

What I'm getting at is that figuring out new test cases is quite difficult (and target program specific)! Critically, the manager program making the decision of "what to test next" has to be able to keep many workers fed with test cases and it has to be able to use the results of those test cases to inform its new test cases. It also has to not consume "too much" compute because any time spent figuring out what to test next could have been used to test something instead! You can imagine a simple server communicating with multiple workers where the workers can generate new test cases and the server sorts them based on a function that calculates the "expected value of bugs found / expected compute required" and sends those tests back to workers to execute. You can further imagine that this function is a neural net that learns based on the results of its tests.

It's unlikely that we're going to come up with some magical algorithm for generating "smart" test cases quickly for our fuzzer – many state of the art fuzzers don't do this. We don't really even plan to try to do that. But what we are going to do is make sure we are able to efficiently communicate the information that could be used to make those decisions between our workers (and potentially a manager server that generates all the new fuzzing tests).

So the general problem for this second axis of parallelism is coordinating the distributed workers into doing useful test cases (not doing work others can do) and then synthesizing their results into figuring out new test cases. I'm speaking about a central "manager" server here but this might end up being fully distributed among the workers.

RESOURCES:

- An (ideally) Nvidia GPU and a CPU with AVX-512 support.
  - If we can only get access to e.g. a Zen 4 APU with its integrated graphics, we can probably work with that.
  - If access to a machine with AVX-512 is a total non-starter, please let us know.
- Sample applications and test cases (FuzzBench and other industry benchmarks may be useful here. Citation: https://github.com/google/fuzzbench)

- We will start from scratch, but like any C++ MPI-based project, we will likely use Boost.MPI instead of the normal OpenMP headers.

GOALS AND DELIVERABLES:

What we plan to achieve:

- A vectorized emulator on the CPU for RISCV32 programs (reasonable programs with small memory footprints, don't rely on clocks, don't make system calls, stuff like that). Ideally one program instance on each vector lane.
- An emulator on the GPU for RISCV32 programs (same reasonable requirements). Ideally one program instance on each thread of the GPU.
- Workers (that use the emulators we've made on the CPU and GPU) that can run test cases (using an input) on a target program and record the branches taken through the code.
- A scalable system for coordinating workers so that they are fed with new test cases to run and can report the results of their test cases to inform what test cases are run in the future. Further, when execution is done this distributed system is going to provide an output indicating which tests it executed and which tests led to unique branches taken through the program. This is going to start out as a system with a manager and workers, where the manager figures out the test cases and collects the results.
- A function that generates new test cases based on results from previous test cases (running either distributed or in a central manager). Again, this function could be extremely complex but ours is going to start simple. A mutation engine that simply doesn't try stuff it's already tried and also prioritizes changing memory locations that it's seen have had an effect will suffice.

What we plan to show off:

- A fuzzer that can take reasonable RV32 programs (likely just toy programs we make) and find bugs in them. We'll likely show some toy programs that might have some insidious bugs and show the test cases that our fuzzer found that exposed the bugs.
- A comparison between our fuzzer and the real fuzzer American Fuzzy Lop for some target programs. Ideally our fuzzer is able to find all the same bugs that AFL is able to find, although we expect that our fuzzer will be slower (AFL was developed with way more resources). Thus, showing that our fuzzer is able to find the same number of unique paths in a reasonable amount of time (say, not 100x slower than AFL) will suffice.
- A demonstration of our two emulators, hopefully showing them scaling really well compared to a RV32 emulator running (non-vectorized) on the CPU when testing many instances at once.

What we hope to achieve if we finish the above:

- A fully distributed fuzzer (no central manager).
- A more intelligent mutation engine for generating new test cases. We'd demonstrate this capability by showing that, using this improved smarter mutation engine, our fuzzer is able to find more unique branch paths per input tested.

PLATFORM CHOICE

- Platform: Linux (it's what clusters and our computers run)
- Languages: C++ and, depending on the hardware we're given, either CUDA or DPC++
    - We know both languages and they're a good fit for high performance code.
- Platform for message passing: OpenMPI (it works well on PSC and is open) with Boost.MPI (for the same reason everyone else uses it on C++ MPI stuff: it wraps MPI with C++ idioms, making for better code, without otherwise changing anything with MPI.)

SCHEDULE:

- Week 1: Have a working emulator for a CPU. Takes an input, records branches taken.
- Week 2: Have CPU and GPU backends
- Week 3: Improving code generation to maximize occupancy and improving communication
- Week 4:
    - First half: Focus on bugs, code polishing, nice-to-haves, etc.
    - Second half: Finish poster, writeup, etc., and get ready to present.