CS527 Term Project: Regression Testing for Software Run on Approximate Hardware

Project Purpose:

Testing approximate software for non-deterministic approximate hardware is different from regular testing in that good tests may show inherently flaky behavior. *I.e.*, an approximate program (run on approximate hardware) may have unacceptable output yet still be bug-free. Conversely, buggy software may produce acceptable results every time it is run on exact hardware, yet when run on aggressive approximate hardware may increase the output error by several orders of magnitude, producing unacceptable results. Therefore, regression tests for approximate computing on unreliable hardware, will likely need to be re-run multiple times in order to generate an output distribution which can be compared to an estimate of what the output distribution should be.

Proposed solution:

Our solution for regression testing approximate programs for non-deterministic hardware is to first characterize a module's *error relationship* of output error compared with injected execution error over a range of different execution error magnitudes. The characterization calculates an ordinary least squares (OLS) regression to represent the module's error relationship. During the next version, the test suite is first run on exact hardware. Any tests that fail would be fixed first. Then when the module passes the tests on exact hardware, the test suite is rerun many times on emulated non-deterministic hardware. This emulation allows use to control the amount of execution error such that we can produce the new version's error relationship. We propose identifying the fraction of these new test results that are not within the 95% confidence interval of the error relationship from the original version. If this is over a threshold of 10% of execution runs, a software bug is suspected.

Results Summary:

We confirmed the observation that commonly used and potentially approximable numerical modules exhibit a strong, continuous relationship between hardware errors encountered during execution and they total output error. We demonstrated that our solution can detect both small and large perturbations to this relationship due to software bugs represented by mutations. Unfortunately, almost all of these detectable bugs are already detectable by traditional tests. Undetectable bugs are a result of equivalent mutants or poor coverage by the original test suite. Finally, we show that our technique may be applicable to numerical functions that are not linear, although a different regression strategy should be used.

Documentation:

1. Clone our GitHub repository
    a. `git clone https://github.com/Harshit661000143/cs527enerj-math.git`

b. This creates a copy of our project workspace with the following directory structure.
    i. docs -- contains this file (final_report.pdf) and all the included figures
    ii. results -- contains all the pickle files (binary python files) for the base execution (<module_name>Gold) and each mutant (<module_name>Muts/<mutant_num>error_value)
    iii. scripts -- our added scripts to generate error relationships and plot results (w describe their uses below)
    iv. enerj -- contains the EnerJ disciplined approximation framework, including our modifications
    v. checker-framework -- the type checking framework EnerJ uses
    vi. checker-runtime -- the runtime corresponding to the type checking framework EnerJ uses
    vii. enerj_apps -- the applications EnerJ was evaluated on
    viii. commons-math3-3.3-src -- the source code for Apache Commons Math
    ix. major -- the Major source code we used to generate mutants

2. Set WORKSPACE environment variable in ~/.bash_profile or ~/.bashrc
    a. `export WORKSPACE=$HOME/Documents/cs527enerj-math/`
    b. `export JSR308=${WORKSPACE}/checker-framework`
    c. `export PATH=${WORKSPACE}/enerj/bin:${PATH}`

3. Creating Infrastructure
    a. `source ~/.bash_profile`
    b. `cd ${WORKSPACE}/checker-runtime`
    c. Execute `ant` command, this builds the checker-runtime library which is required by enerj.
    d. `cd ${WORKSPACE}/enerj`
    e. Execute `ant` command, Building enerj for emulating non-deterministic Hardware.
    f. `cd ${WORKSPACE}/commons-math3-3.3-src`
    g. `mvn compile -DskipTest`

       This compiles all the applications in common-math. It generates the class files which resolves the dependencies for enerj execution.

    h. `cd${WORKSPACE}/commons-math3-3.3-src/src/test/java/org/apache/commons/math3/linear`
    i. `enerjc -cp ${HOME}/.m2/repository/junit/junit/4.11/junit-4.11.jar:${HOME}/.m2/repository/org/hamcrest/hamcrest-core/1.3/hamcres`

```
t-core-1.3.jar:${WORKSPACE}/commons-math3-3.3-src/target/c
lasses/ LUDecompositionTest.java
```

This command compiles LUDecompositionTest.java and generates its class file in the same directory.

4. Identify module (s) to approximate.
    a. Add approximation mark-ups if approximation needs to be constrained. Our default implementation assumes all doubles are approximate. However, if @Approx and @Precise from EnerJ are used, only @Approx data has execution errors injected into it.

5. Suppress JUnit test Failures in order to generate relationship between execution errors and output errors.
    a. Add the following to the end of the JUnit test class file for a given module.

```
public void assertEquals(double obj,double obj1,double tol) {

     try{

           Assert.assertEquals(obj,obj1,tol);

           System.out.println("\n********************PASSED*****
           ***********\nExpected Value:" + obj+"\nActual
           Value:"+obj1+ "\nTolerance:"+tol);

      } catch (AssertionError e){

           System.out.println("\n********************FAILED****
           ************\nExpected Value:"+obj+"\nError
           value:"+obj1);

       }

}
```

6. Run approximation testing framework.
    a. The following invokes the approixmation testing framework for a given module specified in the `cut` variables.

```
$WORKSPACE/scripts/run_major.sh
```

    b. This script automatically performs the following tasks:

i. Generate mutants for approximated module
    1. The following generates the mutants for `${cut}.java`. We specify `${mutant_dir}` where the mutants for that application will be stored. For e.g. In our git repo we have generated mutants for LUDecomposition.java i.e. `${cut}.java` at `${WORKSPACE}/mutants`

       ```
       ${MAJOR_HOME}/bin/javac
       -J-Dmajor.export.mutants=true
       -J-Dmajor.export.directory=${mutant_dir}
       -XMutator="$MAJOR_HOME/mml/all.mml.bin" -cp
       ${commons_classes} $cut.java
       ```

ii. Compiling using Enerj
    1. `enerjc -Alint=mbstatic,simulation -cp ${commons_classes}  ${1}.java`

       Compiling source files `LUDecomposition.java` using enerjc with simulation source-to-source translation enabled that is enabling error injection by enerj.

    2. `mv ${2}/${1}*.class ${4}/`

       This generated `LUDecomposition.class` is being moved to `${WORKSPACE}/commons-math3-3.3-src/target/classes/org/apache/commons/math3/linear`.

    3. For mutants, Hardware error injection is same as on the Base Application.
    4. This generates a class file which simulates Approximate program and allows error injection.

iii. Error injection using Enerj
    1. `cp ${WORKSPACE}/enerjnoiseconsts.json` This file contains all the variables which are used by enerj to decide magnitude of errors.
    2. `sed -i.bu "s/\"DOUBLE_ERROR_MAG\"\:[0-9][0-9]*/\"DOUBLE_ERROR_MAG\"\:${m}/g" enerjnoiseconsts.json`

       We increase or decrease the magnitude of error by modifying the `DOUBLE_ERROR_MAG` between `[0,51]` inclusive. These are the mantissa bits which when higher injects higher magnitude of errors. We inject error to all double opertaions with a probability of `1/10`. We take 10 runs at each Error Magnitude to quantify

randomness in error injection.

3. `enerj org.apache.commons.math3.linear.LUDecompositionTest`

   Enerj is a wrapper which does

   ```
   java -DPrecisionRuntime=enerj.rt.PrecisionRuntimeNoisy -Xbootclasspath/a:$jsr308jar:$cfrtjar:$enerjjar:$plumejar:$target:$junitjar:. org.junit.runner.JUnitCore org.apache.commons.math3.linear.LUDecompositionTest
   ```

   This must be executed from the path `${WORKSPACE}/commons-math3-3.3-src/src/test/java`

9. Run results parsing scripts.
   a. `cp ${WORKSPACE}/scripts/finalErrorExtractor.sh ${WORKSPACE}/resultsL/LUDecomposition`
      This copies all the scripts to the results directory which is required to extract the output execution error and absolute injected error.
      `cd ${WORKSPACE}/resultsL/LUDecomposition`
   b. `./finalErrorExtractor.sh`
      This files does the extraction of all injected errors and execution errors. It stores the output in current directory as `${MutationNumber}error_value` and `${MutationNumber}AbsError_value`

      Following are the details of scripts
   c. `python extractErrorMutants.py`
      This generates the execution output error in a file with name as `${MutationNumber}error_value` in the same directory.
   d. `./extractAbsoluteErrors.sh`
      This generates an intermediate output file `myfile.txt` which has injected absolute errors and is further processed by another file.
   e. `python extractAbsoluteErrors.py`
      This generates the absolute injected error `${MutationNumber}AbsError`
   f. The following converts a pair of text files containing output errors and execution errors into a binary pickle file for plotting and outlier detection. The input files can be in a variety of formats, including one floating point value per line or

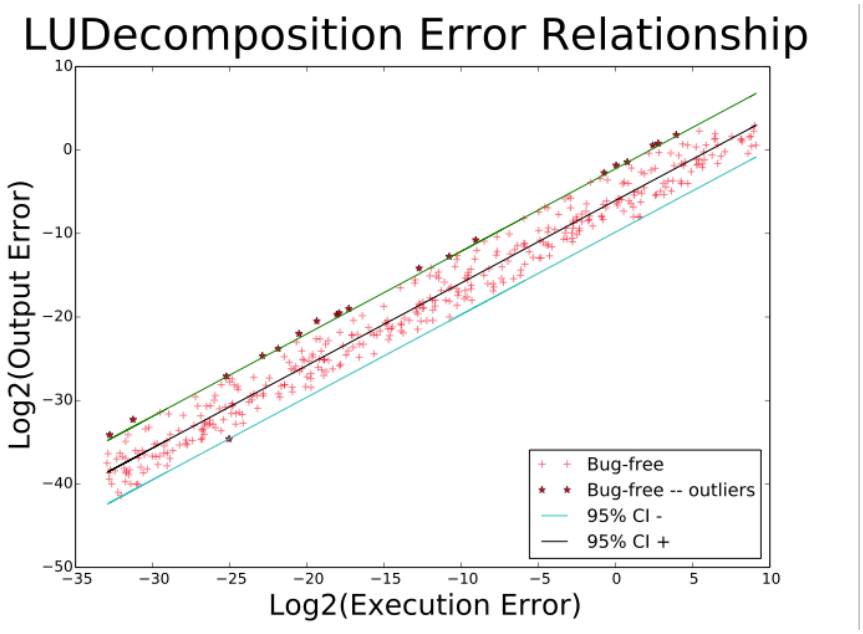
comma delimited on one line beginning with '['.
`$WORKSPACE/scripts/convert_results_single.py -i <path to output error file> -a <path to execution error file> -o <path to output file>`

g. The following converts directories containing pairs of text files containing output errors and execution errors into pickle vectors for plotting.
`$WORKSPACE/scripts/convert_results.py -i <path to directory of output error files> -a <path to directory of execution error files> -o <path to output directory>`

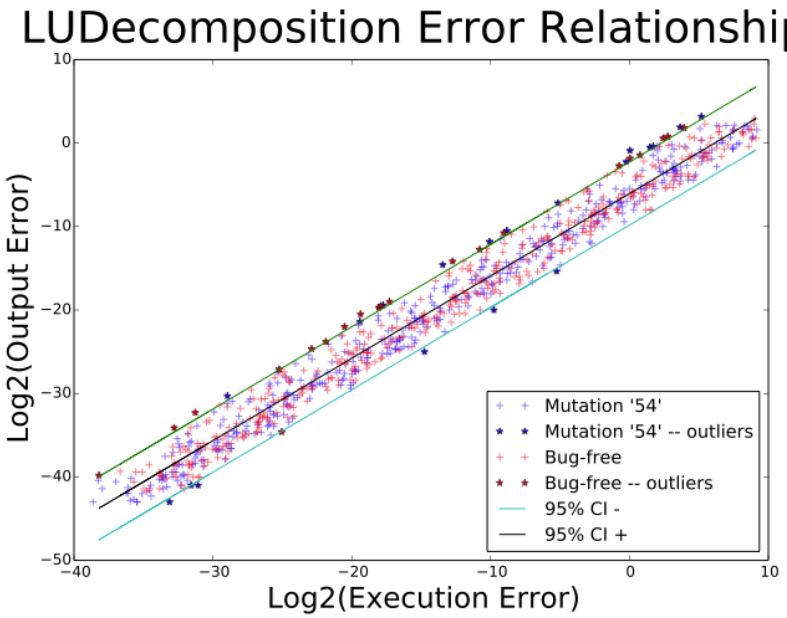
10. Run outlier detection scripts
    b. The following performs outlier detection on single version/mutant and graphically displays the gold distribution, modified distribution, OLS-fitted line, 95% confidence intervals, and outliers. This is very useful for debugging the software testing framework.
    `$WORKSPACE/scripts/plot_inerr_vs_outerr_CI.py -i $WORKSPACE/results/LUDecompositionMuts/1error_value -g results/LUDecompositionGold`
    c. The following performs outlier detection on all versions/mutants within a given directory. At the end it details outliers for each version/mutant, the coverage of mutants (not including the baseline coverage provided by a single exact execution of the original JUnit tests) as well as printing out the versions/mutants that are not covered. This is used to produce the results table below and also quantify the possibility of false positives.
    `$WORKSPACE/scripts/plot_inerr_vs_outerr_CI_dir.py -i $WORKSPACE/results/LUDecompositionMuts/ -g results/LUDecompositionGold`

Results and Tests:

1. We walk through the LUDecomposition as an example of the necessity of and our implementation of regression testing for software designed to run on approximate hardware.
    a. Our first observation is that there exists a relationship between execution error from approximate hardware and output error in the absence of software bugs. The following figure shows this relationship summed across all tests contained in LUDecompositionTest from Apache Commons Math.

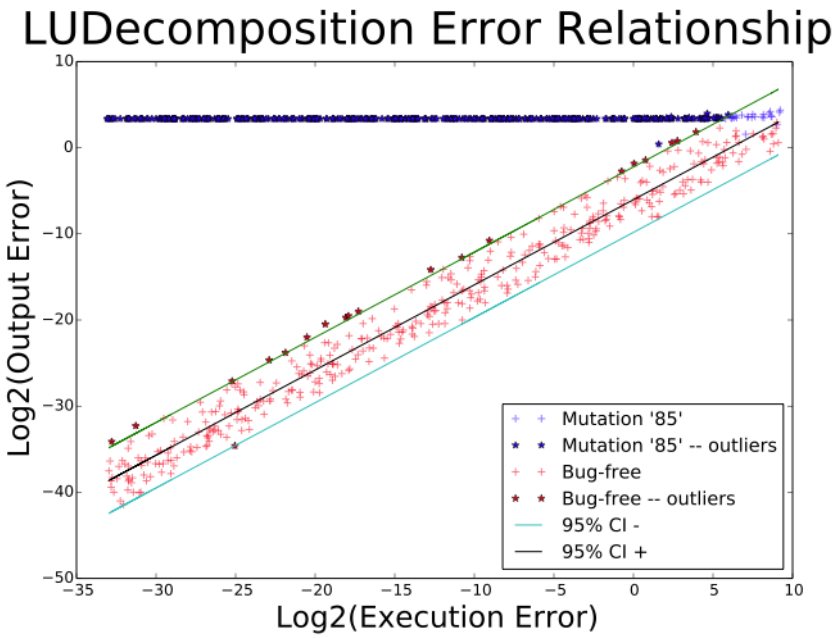


b. When a software bug (as represented by mutants) is inserted, there may be one of four general responses.
    i. The first response is where there is no change in the relationship between execution error and output error. This can occur when a given mutant is not exercised by the existing tests or when the mutant is an equivalent mutant. These are either not correctness bugs (equivalent mutants) or require additional tests to identify. Below is an example of an equivalent mutant where a mutated initialization value is always overwritten before it is used (mutant 54).

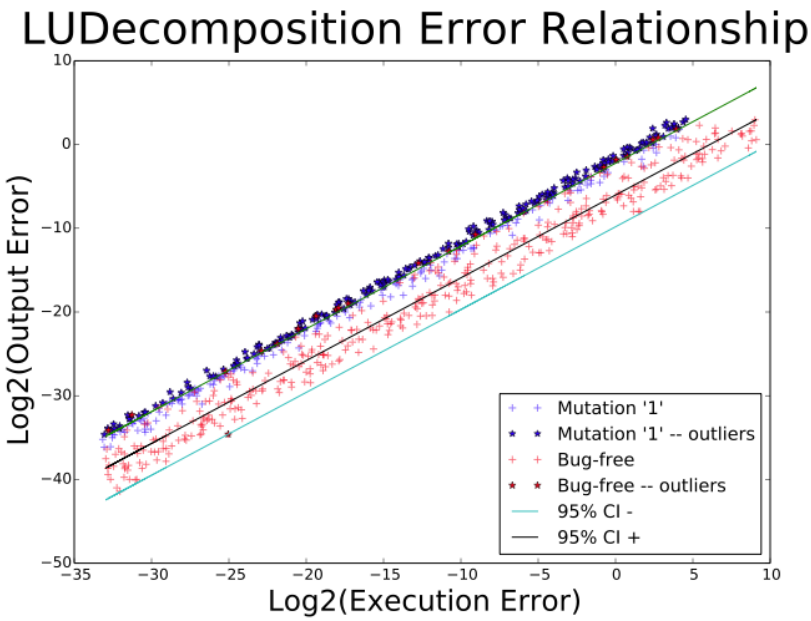


ii. The second response is where there is an assertion failure even on exact hardware. Mutation 4 is an example of an array out of bounds exception due to changing a for loop exit condition. These bugs do not require our framework as they are already caught through traditional testing.

iii. The third response is where the relationship is dramatically changed. Below is an example where a line copying the result of an accessor method is removed (mutation 85). There is a large constant error from the software bug which overshadows the execution errors. This type of error is generally caught by testing on exact hardware.



iv. The fourth and final response is where the relationship is slightly changed due to the software bug. For example, the figure below shows the case where an initialization value is mutated slightly. The software bug increases the output error by several orders of magnitude for any given execution error. When the amount of execution error is low or non-existent, this may be acceptable or even undetectable as is the case when run on exact hardware. However, this bug would seriously limit the output quality when run on more aggressive approximate hardware. These are the bugs we aim to detect.

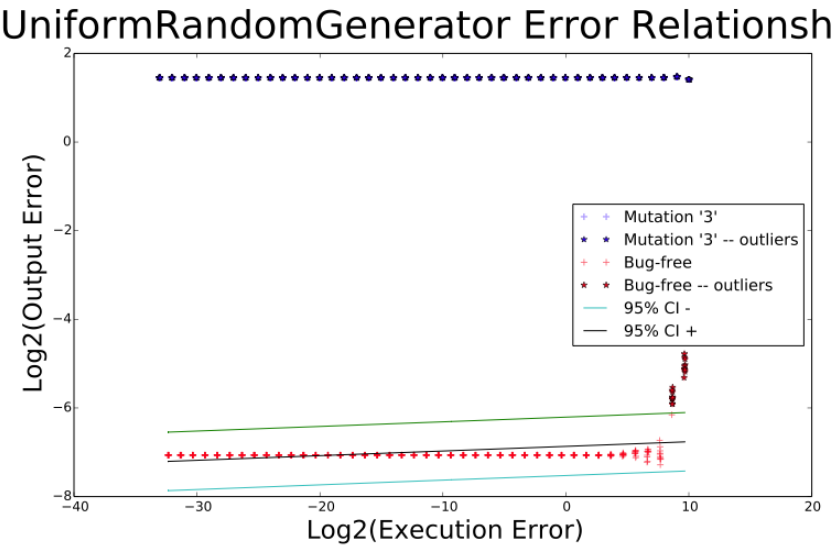


c. For LUDecomposition, our technique caught 43 out of 233 mutants compared with 68 caught through traditional (*i.e.*, exact) testing. Three of these mutants are unique to our technique. Through manual inspection we identify that the remaining mutants the traditional testing caught that were not caught by our technique were due to exceptions during the test execution. Currently, in our implementation these manifest themselves as having no outliers because they have no points. Obviously in a real implementation we wouldn't perform our technique on tests that crash for exact hardware.
d. One method we used to test and validate our technique was to perform multiple characterizations of the base version (no mutants) and compare these emulations to each other. We generated 22 characterizations for the base code and our technique correctly identified them as containing no software bugs. In fact, we confirmed that any of these baselines would produce the same results when compared with mutants.
e. Another method we used to validate our scripts was to manually identify equivalent mutants and check that our technique could not detect them. Our final implementation does not detect any of the equivalent mutants we identified (*e.g.*, mutant 54 shown above).
f. One significant overhead of our technique when applied to a test suite is the substantial increase in execution time used to generate a statistical error relationship (*e.g.*, running the test suite 520 times per version of a module). Therefore, we explored selecting fewer runs during testing for new versions/mutants.
    i. First we chose to eliminate the lower error regions which tend to be

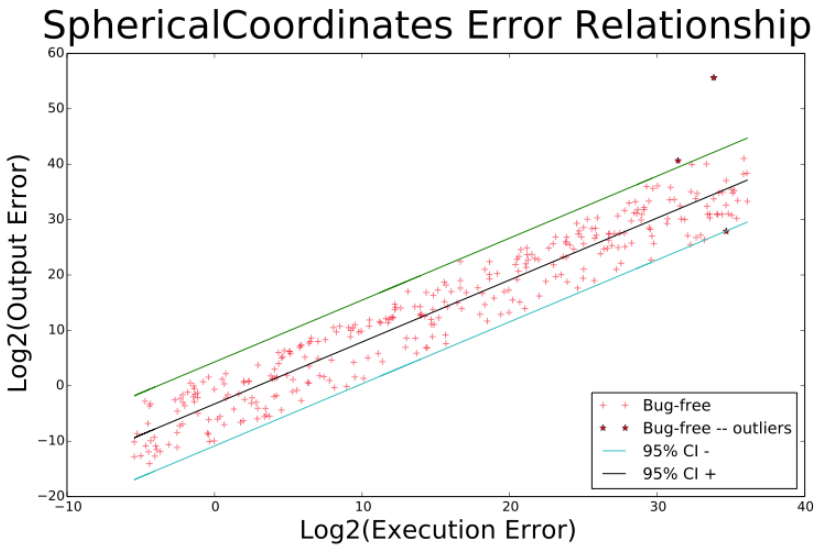
more noisy and less regular. We could eliminate all errors below 0.1 with no loss in precision while reducing the test time by over 75% (estimated based on # of runs). Eliminating all errors below 1 resulted in a coverage loss of one mutant and an execution time reduction of 85%. Eliminating all errors below 10 resulted in a coverage loss of five mutants and an execution time reduction of 97%. This is to be expected as execution errors begin to have the same order of magnitude output error for software bugs of type iii (described above). We did not detect false positives for using the upper range.

ii. Second, we chose to only select one or two points from each magnitude of execution error. However, this resulted in an increase in false positives of twelve and three mutants, respectively.

2. To evaluate the generality of our technique, we applied it to Apache Commons Math modules not in the linear category.
    a. UniformRandomGenerator
        i. UniformRandomGenerator produces an identifiable error relationship, although it would appear to be more non-linear. Despite this non-linearity, our technique can still identify all 18 mutants because each mutant produces a significant, large magnitude error (mutant 3 is shown below). The traditional correction can also detect these mutants.



    b. SphericalCoordinates
        i. SphericalCoordinates produces a linear relationship as shown below.



c. KolmogorovSmirnovDistriubtuion
    i. We don't show results for KolmogrovSmirnovDistribution as it didn't show any relationship between the injected error and output error. It had significant amount of noise distributed throughout the plot and therefore distinguishing anomalous distributions would be difficult without more sophisticated techniques.
d. Prime
    i. This application was slightly confusing to us. Even with injection of larger magnitude of errors we didn't see any absolute error injected nor the execution error. After looking into the code we saw it had no double operations, all the data types used in this application were either int or boolean.