Emily Johnson

Dependable Systems Project Final Report

All code used and collected data can be found in this repo: <https://github.com/jopagu/DS_Project>

Intro:

This project is based around testing the resiliency of various programming languages and compilers. The goal is twofold: I wish to test whether more complex languages and language features have an impact on a program’s rate of harmful soft errors, and I secondarily wish to test the importance of the actual compiler for determining a program’s resilience. When developing safety critical applications, such as autonomous vehicles, every little bit of resilience can save lives. As such, it is important for developers to be cognizant of how their choices during development can impact the dependability of the final project. I wish to determine whether something so simple as choice of language or compiler could be an important consideration.

Background:

When a soft error occurs, a random bit will be flipped. In many cases, this doesn’t affect the final output of the program, in which case the error is considered benign. Otherwise, it may cause the program to crash, hang, or output an incorrect result, which is known as a silent data corruption (SDC). We wish to reduce the occurrence of all three of these negative results, but SDCs are the most dangerous, as they can pass by undetected.

To this end, I am utilizing the compiler infrastructure suite LLVM. LLVM has a consistent intermediate representation that can be taken advantage of by any language so long as someone has built an LLVM compiler for it. From LLVM’s website, “Through these external projects you can use LLVM to compile Ruby, Python, Haskell, Rust, D, PHP, Pure, Lua, Julia, and a number of other languages”. The versatility of LLVM is perfect for performing consistent testing across several languages.

In conjunction with LLVM, I am using LLFI, a fault injection tool that acts on LLVM code. Fault injection is a standard method for testing program resilience and is able to inject bit flips at random or at specific locations in a program. Ultimately, it can determine whether the injected fault caused an SDC, a crash, a halt, or was benign.

Methodology:

I am taking advantage of LLVM’s wide reaching language support to use LLFI to test the resilience of programs in many languages. By implementing the same algorithm in each desired language, I can then, in theory, compile them to LLVM and perform fault injections to see if the resulting code is more resilient based on the initial language.

Each of my fault injection campaigns will run the program 10,000 times, with one random bit-flip per run. The bit-flips are able to target any instruction at random. The run is considered a hang if it goes for more than two seconds, which is reasonable based on the short runtime of my chosen algorithms.

The first program I implemented is a Taylor series approximation for square roots. The C version of this algorithm is included in LLFI. This algorithm utilizes basic, common, language features (if else, for loops, function calls), making it desirable to test. Additionally, the simplicity of the algorithm means that the code will look similar in each language, making this a useful control test.

The second program I implemented is a binary search on an array/list. I thought this algorithm would be useful to test the reliability differences between more and less complex language features. With this algorithm, I could test different languages’ implementations of common data structures, e.g. C’s array vs. C++’s vector. Unfortunately, using this many memory access instructions led to too many crashes from fault injections, giving no valuable data on SDCs. The further programs I wished to test were similarly reliant on data structures, so I decided to test no further algorithms for this project. Additionally, I was unable to obtain C++ results for this algorithm, as for unknown reasons LLFI did not produce the file “llfi.stat.prof.txt” which is essential to performing the fault injections.

The languages I was able to successfully test were C and C++, using Clang and Clang++ as compilers. I used Clang/Clang++ version 3.4, which is the same version of LLVM that LLFI is built for. This is because Clang is built directly with LLVM, and shipped together with it, making the versions the same. I additionally attempted to test Haskell and Rust, but I was unsuccessful. While both of these languages have LLVM compilers, and were able to be successfully compiled to LLVM IR, these compilers are built with newer versions of LLVM than 3.4. Due to this, LLFI was unable to run on them, producing errors consistent with C/C++ code compiled using a newer version of Clang.

I attempted to work around these issues using an LLVM C backend (found [here](https://github.com/JuliaComputingOSS/llvm-cbe)). This tool promises the ability to compile LLVM IR code into C. I had hoped that I would be able to take LLVM from another language, turn it into C code, and make that into LLVM code that worked with LLFI. However, this tool proved unsuccessful at these other languages, possibly due to version issues again. I was only able to get it working on LLVM code that was originally C. However, this still provided valuable data on the impact of compiler choice for resilience.

Results:

For the square root program, the results are very similar between C and C++, which is in line with what I expected for these languages. C++ is built on C, and the only differences in my algorithm are quality of life language features. Additionally, I was using the clang compiler for both, likely leading to similar compiled code. However, I was surprised to see that the C++ program was consistently having lower rates of malign faults. The data shows that C++ was approximately 1% more resilient. This is potentially due to the fact that C has less memory safety, meaning that a soft error could more easily cause severe issues in the program memory. In contrast to this similarity, the C code recompiled with cbe performed significantly worse, with a much higher likelihood of crashing. This is to be expected, as this code involved running 3 different compiler phases (C -> LLVM -> C -> LLVM). This results in code that is likely extremely unorganized and unoptimized. This result shows proof that compiler choice does matter, and that more optimized compilers can produce more resilient code.

With the binary search program, I was unable to test C++ code, leaving only a comparison between C and C with cbe. This program resulted in a significant amount of crashing, with few SDCSs. This is possibly due to the number of memory access operations, meaning that an error will cause an improper memory access crash, rather than data corruption. However, the results with cbe were extremely surprising. The program ended up significantly more likely to have a benign error. However, this resilience comes at a trade-off, as this compilation introduces a noticeable rate of SDCs. I have few ideas about why this unoptimal compiler chain causes more resilience, but I can only assume that the recompilation somehow adds a higher degree of memory safety.

Conclusion:

While many of my initial plans for this project ultimately failed, it still produced some interesting results. First, we see that C++ proves more resilient than the lower-level C for simple applications. Additionally, more complex compilation will produce programs that are less safe. However, we see that with programs requiring many memory accesses, the more complex compilation actually produced more resilient code, but with harder to detect errors due to the SDC rate. Ultimately, the hypothesis that language and compiler choice can play a role in program resilience was proven true. Further work should prioritize producing a more versatile version of LLFI, so that additional languages can be successfully tested. Additionally, further algorithms should be considered, especially if the memory access issue can be worked around.