ECE 455 - Fault Tolerance Mechanisms

Learning Objectives

The goal of this lab is to implement four distinct fault tolerance mechanisms.

After successfully completing this lab...

- You will be familiar with some of the techniques used in safety-critical systems to mitigate the effect of faults or recover from them in order to prevent failures in the system.
- You will understand how the various types of redundancy and diversity of both operations and storage in embedded software can provide fault tolerance.
- You will have the necessary skills to implement these techniques using the C programming language.

Background

Embedded systems are often used in safety-critical applications. Hardware failures in the processor or memory can cause catastrophic failure. To mitigate these risks, we implement a number of different fault-tolerance mechanisms.

What You Will Be Doing

In this lab you will implement a simple algorithm to find square roots of double-precision floating point numbers using the Newton-Raphson method (described in the appendix to in this manual). To ensure the correctness of the results, you will use four different fault-tolerance mechanisms:

1. **Data redundancy:** You will set up a mechanism to store data in duplicate form, with one of the “clones” stored with the bits inverted. The inverted duplicate facilitates early detection of stuck-at faults — a bit in the hardware may be stuck at some particular value, and this fault could go undetected (and potentially lead to an error and failure at a later time) if the value being accessed happens to have the same value at the faulty bit. A suggested way to implement this functionality in C is setting up a user-defined type (a struct) for each data type (for the purpose of this lab, we will only deal with int and double) that contains two data members: the value and the inverted value. You will also provide a function to read the value and a function to set the value. The read function will return a success code indicating whether a fault was detected and will write the retrieved value in a parameter passed by reference.
2. **Voting system:** Several types of voting systems exist, both for storage and for computations. For the purpose of this lab, you will implement a majority voting system for a redundantly computed value. You will execute your sqrt function three times in succession, and accept the result if at least two of the computed results agree. Since we’re using double-precision, “agreement” between two values means that the two

values are within a given distance ϵ. For the purpose of this lab, we will consider $\epsilon = 10^{-6}$ (1e-6 in C/C++). In other words, agreement between x1 and x2 means that $| x1 - x2 | < 10^{-6}$)

3. **Heterogeneous computations:** This technique is similar to the previous one, but it introduces diversity -- each of the three computations is executed with a different, independently implemented algorithm. In our case, we will compute result #1 using the C standard library `sqrt` function, result #2 using the Newton-Raphson method, and result #3 using binary search to narrow down the result to within 10^{-6} accuracy. The voting/decision protocol in this case is:
    a. if result #1 agrees with either one of the other results, then return result #1
    b. if result #1 disagrees but results #2 and #3 agree, then return result #2
    c. if all three values disagree, inform client code that a fault occurred (to this end, you could use a mechanism similar to the one described in item 1).
4. **Result verification:** With this technique, we verify the result of a (possibly complex and thus potentially error-prone) computation by applying the inverse operation or some result confirmation operation. In our case, since we're computing square roots, we verify the value returned by that function by computing its square and checking whether it agrees with the input value (the one for which we computed the square root). Again, "agree" means within 10^{-6}.

Simulating Errors

In order to simulate errors, you will be provided with a module that you will use to access data. That module provides you with the following functions:

```
void fault_injection_reset(void);
int faulty_int(int original_value, int fault_type);
double faulty_double double original_value, int fault_type);
```

The functions `faulty_int()` and `faulty_double()` receive a pointer to an int or double (respectively) and a flag indicating the type of fault (`RANDOM_FAULT` or `STUCK_AT_FAULT`), and return the value in that variable with the possibility of a fault.

For `RANDOM_FAULT` type of faults, the functions `faulty_int()` and `faulty_double()` will randomly inject faults, but no more than one fault. Calling `fault_injection_reset()` clears the condition that a fault has been injected, so that the functions will again inject faults.

For `STUCK_AT_FAULT` type of faults, the functions will always return a faulty value, with the same (single) bit being stuck at a given value. The functions will randomly choose the bit and the value for that bit (independently selected for faulty ints and faulty doubles). Calling `fault_injection_reset()` clears those selections so that the next call to `faulty_int()` or `faulty_double()` will choose a new position and a new value for the faulty bit.

You should be careful where you call these functions. Notice that a fault in intermediate values in iterative methods such as Newton-Raphson can cause the execution time to exceed what we assumed would be the worst case execution time following a mathematical/theoretical analysis. In particular, notice that in a real-life application one must take this into account, since a random hardware fault can happen *anywhere* in our code and at *any time* during execution.

Submissions

You will submit a brief report including the source code for the TA to grade. The report should briefly discuss the types of faults (systematic software faults, hardware faults, single-bit, random, stuck-at, etc) that each of the implemented methods are effective against, as well as those for which the method is not effective. This should be an ordinary text file called “report.txt”.

You should put your C source code in a file with the following name:

your ID number`-ece455lab3.c`

For example, “`20171234-ece455lab3.c`”.

You may have additional C source files in the same directory.

Put all your code and your report into a compressed tar file with the following name:

your UW userid`-ece455lab3.tar.gz`

For example, “`broehl-ece455lab3.tar.gz`”

If you do not follow the submission instructions, you may be penalized up to 5 marks.

Appendix -- The Newton-Raphson Method

A reasonably efficient way to compute the square root a for a given value *a* is to use the Newton-Raphson method to solve the equation $x^2 - a = 0$. You can look up the details online (or in your course notes from your math or numerical methods courses), but the “in a nutshell” version is as follows....

The iteration step is

$$x_{k+1} = 1/2 \ (x_k + a / x_k)$$

The iterative process stops when we are within a specified distance ϵ from the correct value (or alternatively, since we are solving an equation of the form $f(x) = 0$, we stop at iteration $n : |f(x_n)| < \epsilon$).

The choice of initial value is non-problematic: any initial value $x_0 > 0$ guarantees convergence (albeit potentially slow convergence). You can choose $x_0 = a$ as a simple way to obtain reasonably good performance.

Tip: There's an interesting bug that can be useful for this lab. Instead of stopping when the result is close enough, hardcode a fixed number of iterations. For large enough input values, the number of iterations will not be sufficient to get close enough to the result, yet for other, smaller input values, the result will be correct.