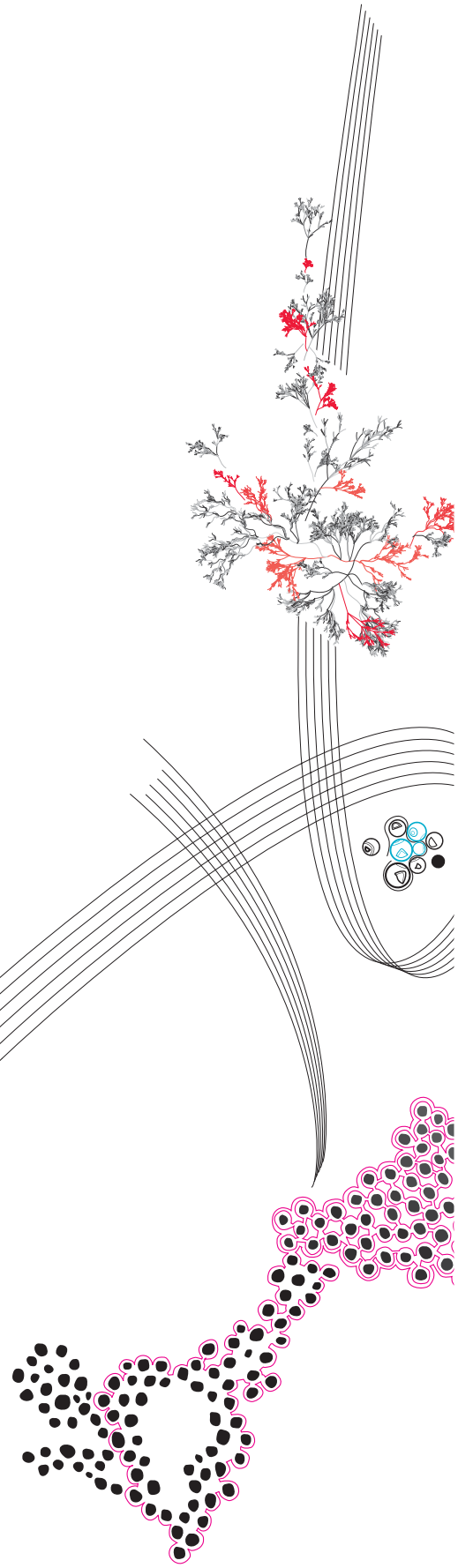


BSc Thesis Electrical Engineering

Reliability analysis of Triple Modular Redundancy

Stijn Nowee

Supervisors: B.Endres Forlin, M.Ottavi

July, 2022

Department of Electrical Engineering
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Abstract

In this paper the effect of a Triple Modular Redundancy(TMR) implementation on the reliability of a system is examined. To accomplish this, a micro-processor with a RISC-V architecture has been simulated with and without the TMR implementation. In the simulation are both Single Event Transients(SET) and Multiple Event Transients(MET) injected. Additionally, a transistor fault has been simulated with the TMR implementation. The TMR is applied to the Multi/Div block of the processor and the faults will be injected at the input of these triplicated blocks. The performance of the systems with and without TMR will be compared using the ratio of the number of faults injected to the number of faults propagated. When the system is only injected with SET's, the system without TMR has a ratio from 0.058 to 0.389 depending on the probability of a SET occurring, while the system with TMR does not propagate any fault at all. If MET's are injected the system without TMR performs better with a ratio between 0.069 and 0.291, while the system with TMR has a ratio between 0 and 0.036. The TMR implementation reduces the probability of an error propagating significantly, but if a Multiple Event Transient hits multiple similar wires, it can still fail. To combat this other forms of redundancy should be implemented.

Contents

1. Introduction

An embedded system is a combination of a processor, memory and input/output peripheral devices, where it has a dedicated function within a larger mechanical or electronic system [6]. Because these micro-controllers have a dedicated function, they can be optimized for a specific task. In the field of embedded systems, efficiency is extremely important. The efficiency is so important because of the specific implementation of these micro-controllers. Often, these embedded systems can be found in small devices, or wireless devices that run on battery power or a number of other use cases. Because of this, some embedded systems should be optimized for maximum performance, while others might be constrained by their power consumption or limited by the physical space they have available to them. One of the ways that these goals can be achieved is by removing redundant parts from the system. But over the years, redundant components have been used in many places in almost every computing system. For instance, in the aviation industry it is very common to see multiple types of redundancy implemented. The Boeing 777 uses a triple modular redundancy scheme of it's computing system, airplane electrical power, hydraulic power and communication path [12]. Another example is the recently launched James Webb telescope. This telescope has a lot of different systems, many with redundant components, such as it's "Hemispherical Resonator Gyros". Internally these include two processors and power supply boards of which one pair is redundant. They also include 4 internal gyro's of which 3 are redundant[5]. In the aviation and space industry, weight is one of the most important factors, so the weight of every system should be as low as possible. Thus this seems counter intuitive. The reason given for implementing these redundant systems is that it would improve reliability which is a major concern when it comes to human life or space projects that cannot be repaired. This results in the following question: "How does redundancy improve reliability?". To answer this question, we made a simulation of a micro-processor and implemented a redundancy scheme on it. We will target the aviation industry where the application is time critical thus the redundancy scheme should reflect this. The implementation will be tested by injecting the simulated micro-processor with multiple bit flips and monitor the output to test the reliability of the system.

2. Theory

2.1 Processor

To implement a redundancy scheme, a customizable processor is needed. In this case, we chose the Ibex RISC-V Core [8]. The core is heavily parametrizable which makes it perfect for this application. A general overview of the core can be seen in fig. 1. Because the specific specification of the module aren't that important for this research, we chose to use a bare metal setup. To test the redundancy scheme, two different versions are made. One version with the redundancy scheme implemented, an one without the redundancy to compare the results with. The multiple variation will be discussed in section 2.3. Both versions will be based on the ibex "Simple system". The simple system is an Ibex based system that is targeted for simulation. To actually simulate these variations, verilator is used. More on this simulator in section 2.2. The ibex core consist of a two-stage pipeline. The first stage is the Instruction Fetch (IF) stage. This stage fetches the instruction from memory and is capable of doing this with an IPC (Instruction per Cycle) of 1. The second stage is the Instruction Decode and Execute (ID/EX) stage. The fault injection will take place in this location. More specifically, it will take place in the multiplier and division block contained within the execution block. This has been chosen because the use of flip-flops that are able to be influenced by the fault injection and the potential of the errors propagating and cascading into multiple errors. This can be analysed and used to see whether a redundancy scheme is effective enough to compensate for this behaviour.

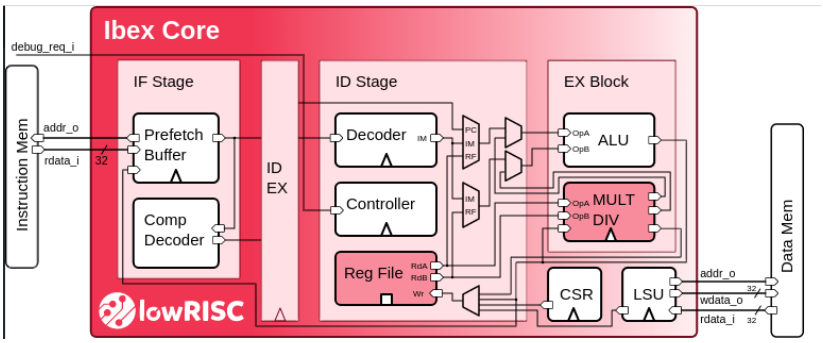


Figure 1: Ibex core block diagram

2.2 Simulation

To simulate the ibex core processor, Verilator is used in combination with Gtkwave. Gtkwave is used to generate wave-forms and it is neatly integrated into Verilator, it only requires an extra flag to work. There are a few big reasons why we chose to use verilator. Verilator is not a traditional simulator, but it acts more like a compiler. It converts the SystemVerilog files from the ibex processor into a C++ model, after which this model can be executed. Because

this runs compiled code, this simulation is extremely fast when compared to other simulators. According to verilator[11], if the simulator is running on a single thread, verilator is about 100x faster than interpreted Verilog simulators. When multi-threading this can increase with another 2-10x times speed increase, resulting in a 200-1000x speed improvement over interpreted simulators. Secondly, Verilator has a Verification Procedural Interface (VPI) which will be very important. We will use this interface to inject faults into the simulation, more on how exactly that is implemented in section 2.4. The VPI is able to directly access the public signals of the processor. These signals can be read from and written to by using the corresponding functions provided by the library. Using this interface, faults can be inserted at the correct time at the correct place. The final reason why verilator was chosen as a suitable simulator for this research is linked to the previous advantage. Verilator is highly customizable using c++ code. Due to this the system for injecting faults can be implemented by editing the verilator simulator. Unfortunately, Verilator also comes with a few disadvantages. Because verilator is not a traditional simulator, it can conflict with injecting faults. The way verilator works is different from an event based simulator. Values that are changed by the VPI will only propagate when the whole system is evaluated, by calling the *eval*() function. This can only happen once per clock cycle. This means that reading a value, editing this value and writing it back to the signal cannot happen in one clock cycle. This will be further elaborated on in 2.4. Furthermore, because verilator acts more like a compiler, it can optimize certain parts of code if it concludes that these have no influence on the program or will always be true/false. Verilator does not take into account that some of these values might actually be changed by the VPI and thus shouldn't be optimized away. There are workarounds for this issue that have been implemented. But together with the previous issue these are one of the biggest short comings of Verilator when using it for fault injection.

2.3 Redundancy Implementation

2.3.1 Redundancy schemes

There are currently many different redundancy schemes[7]. These can be categorised in four main categories, information, time, software and hardware redundancy. Information redundancy in general is implemented by appending additional bits to stored or transmitted data. The additional bits are generated in a certain pattern such that it can convey information about the block of bits that have been sent. A well known example of this is the hamming codes. The additional bits can be used by the receiver to detect if there were errors and is able to correct them. Of course, different types of information redundancy schemes have different capabilities and different levels of overhead. The second category is time redundancy, time redundancy often doesn't need a change in hardware, which can be very valuable when physical space is limited. Unfortunately this does come at different cost. It works by having the system perform the same task multiple times. It then compares the output of the same task and can correct for any errors that possibly have occurred. This comes with a substantial execution time cost. As with the information redundancy,

there are many different ways of implementing this and each have their own positives and drawbacks. Next are software redundancy schemes, these are mainly used against software failures. This targets man made errors (bugs) and tries to avoid them by having for instance multiple teams work on the same functionality but separate from each other. These two different software implementations are then used side by side. When one encounters a bug it is unlikely that both software programs have the same bugs and thus can prevent the issue. This however are not the faults that we are targeting in this research thus will not be looked at further. Finally, there is the hardware redundancy, in general this redundancy category copies certain parts of the processor. After these copies a voter block is placed to decide the correct outcome. A common example of hardware redundancy is Triple Modular Redundancy or TMR in short. Because we want our application to have a high execution speed and the program running on the system will be time critical, the time redundancy will not be suitable for this system. To be able to correct for multiple faults injected at the same time, hardware redundancy will be a good choice. A deeper dive into the specifics of the chosen hardware redundancy can be found in section 2.3.2. These different types of redundancy schemes can all be used at the same time but because hardware redundancy and time redundancy both cover the same type of errors it is uncommon to see both of them combined. Often only one of these redundancy methods is used in combination with information redundancy.

2.3.2 Triple Modular Redundancy

TMR is a method where a certain module is triplicated and a voter block is placed behind it. For this research this redundancy scheme will be applied to the Multi/Div block in the EX stage of the ibex controller (section 2.1). This module has been chosen because this module is actively being used by the benchmark that we will run on the processor. It also provides a possibility of the error propagating through the module before it hits the voter block. This could make it harder for the TMR to error correct, but will result in better "Worst case scenario" results. A very simple overview of a TMR system can be found in fig. 2. The true implementation however, is a little bit more complicated. This is because instead of one output, the multi/div block has 6 outputs that need to be connected to voters. This means that there should be either one very big voter block that can account for all 18 inputs and 6 outputs, or many small ones that all take in 3 inputs and 1 output, or something in between. In this case many small voter blocks were chosen for multiple reasons. First, it provides better flexibility, these voter blocks are not depended on the block that came before, in this case the multi/div block. Instead these can be more easily relocated and implemented compared to the big voter blocks that can only be used in one. Secondly it allows for a more organized structure and changes are more easily made. However, there are different types of voter blocks. For this setup a standard majority voter is used. This voter block is works by receiving three different inputs from the three duplicate blocks. Then compares these inputs bit by bit, this is needed when arrays are passed. When two signals are identical, it duplicates the value of one of the inputs to a buffer, which is assigned to the output. If all three signals are different, the voter block will

choose an input at random as it's output. In the case of these specific blocks, input three is always chosen. Of course, this voter block only works for TMR, if the system instead of three, has five duplicates, the voter block should be adjusted accordingly and compare five inputs.

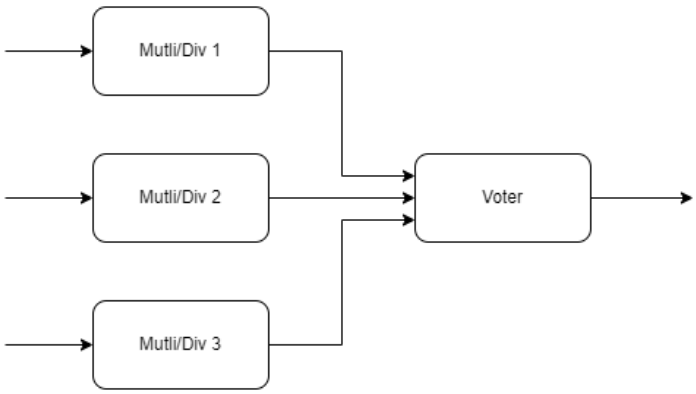


Figure 2: Simple Triple Modular Redundancy overview

2.3.3 Single Event Effects

As with everything in computing, there are trade-offs that need to be made. This is no different selecting which parts will be triplicated. Even though we are not constrained by physical space, cost or performance goals, to keep the results relevant for other use-cases not everything will be triplicated. This would be a huge investment for extra hardware. The decision on where the triplication takes place depends on the type of error it should filter out. Single Event Effects (SEE's) are caused by charged particles that hit electronic components. This can result in multiple different effects which can be divided into two categories, soft errors and hard errors. In computing, a soft error is a type of error where a signal is incorrect, however these errors do not imply a mistake in design or construction[2]. If a soft error is observed, there is no implication that the system is any less reliable than before. Due to this fact, soft errors do not persist over a power cycle. These soft errors can be caused by neutrons from cosmic rays or alpha particles from packaging material. It used to be the case that soft errors were only a major concern for space applications [9]. Both because reliability is very important as these space projects often could not be repaired if anything went wrong, but also because of the increase in exposure to cosmic radiation, making soft errors more likely to occur. Furthermore as technology progressed and the transistors got smaller and smaller, the probability that one of these bits got flipped by a radiation particle grew. This is because smaller transistors are less resilient against charged particles hitting them. Soft errors can be caused by two different events, Single Event Upsets and Single Event Transients. A Single Event Upset (SEU) is an event where a charged particle hits a memory bit and flips it. This bit will stay in memory until it is overwritten. This means that it can cause problems when this memory bit is being accessed. The second event that can cause a soft error

is the Single Event Transient (SET). This event occurs when a charged particle hits combinational logic and flips a bit in a wire for a very short time. It creates a peak or negative peak that will decay over time. If this peak lines up with the use of this signal in the combinational logic it can cause a soft error. Please note, such a SET could transform into a SEU if this fault is propagated through the combinational logic and stored in a register. The second category is hard errors. These, in contrast to soft errors, are destructive. A Single Event Latch-up (SEL) is such a hard error. This event can cause an apparent short-circuit. Without proper countermeasures in place, this error often destroys the device due to thermal runaway[1]. Because we are planning on simulating the Event Effects injected, this type of error will not be used.

The type of event that occurs in a certain block is dependent on the composition of that block. If a block contains many flip flops, it is more likely that there will be many more SEU's with respect to SET's. Thus to know what would be realistic for the ibex system that we have. We need to look at what the composition is of the system. Because the Ibex simple system only consist out of a pipeline of two stages (see section 2.1, it has very few flip flops. Most blocks in the system do have flip flops but these are barely used and most of the processes work combinational. Because of this, implementing SET's is more realistic and an addition of SEU would not yield different results. There are of course flip flops in the system registers but writing faults to these will result in an early termination of the system because it cannot function with faults in these locations. This would make gathering results much harder and thus not feasible.

2.4 Fault injection

The fault injection is assisted by special fault injection blocks. These are placed before the triplicated blocks on all of the inputs that need to be manipulated. A more complete overview of the system can be found in fig. 3. The fault injection blocks are very small, it only consist of a single XOR gate per input signal. The each XOR gate is connected to a input signal that normally would have connected straight to the multi/div block and with a dummy signal that is zero by default. This dummy signal is used to actually execute the bit flip. The decision on when and where to flip a bit will be done by the error generator which will be explained in 2.4.1. If the dummy signal is set to something other than a zero, the manipulated signal is send to the corresponding Multi/Div block. Naturally, if there is no fault scheduled the dummy signal will be zero, there will be no manipulation done and the input is directly propagated to the output. As stated before, the injection takes place before the triplicated blocks to simulate an Single Event Transient. This is easier to implement and to upscale compared to injecting the faults into the Multi/Div themselves. The bit flip can be implemented in two ways. The first involves using the Verification Procedural Interface (VPI). Using this method a value can be read, manipulated and be written back to the same location. This method however can run into some issues. To read a value of the current cycle, the system first needs to be evaluated, otherwise a value from the previous cycle will be read. After evaluating, the correct value can be read out for that clock cycle and

manipulated. Unfortunately, attempts to make the system evaluate again (In the same clock cycle) were unsuccessful. This means that while the modified value was written to the correct signal, it was not propagated. This again could be more easily done on a more traditional simulator. The second option which is the one that is implemented here, in contrast to the first option, extra injection blocks need to be implemented for this method to work. This is very unfortunate, because this means that the actual structure of the processor has to be changed for the injection of faults. This makes the representation of the real world slightly less accurate because of the extra components needed. The fault generator chooses a bit to be switched. The location of this bit is passed to the dummy signal of the injection block using the VPI. Here the bit flip is applied by simply having an XOR operation on the input with the bit location. During evaluation this dummy signal is taken into account and thus the bit flip is applied. After a half cycle, the dummy variable is reset to zero and thus the bit flip is reversed. This emulates the behaviour of an SET which wouldn't be saved in memory.

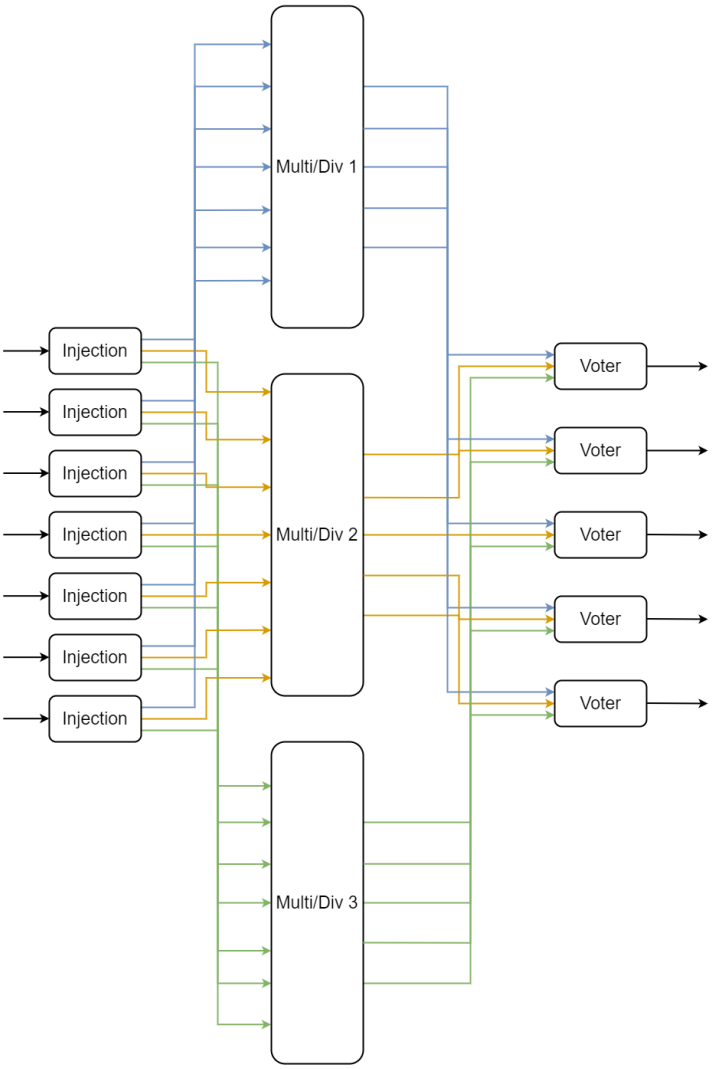


Figure 3: Fault injection setup with TMR on Multi/Div blocks

2.4.1 Error generation

The error generation is not a trivial part of the fault injection. These errors should be inserted in a realistic manner to receive realistic results. The main source of bit flips is radiation, so it would make sense to let the error generation be related to the amount of radiation the processor receives. While this is true, there are a lot of different processors with different types of protection, such as a thicker casing around the processor. Radiation applied from outside of the processor does not always have the same influence on the number of bit-flips that occur in the system. This resulted in choosing for a more general measurement: The probability of a bit-flip occurring for each bit in a cycle. Every bit that can flip, thus all the bits going trough the injection blocks, have

the same probability to flip. The script for deciding which bits will be flipped and when was written in Python. The results were stored in a csv file, the information contains the time when this fault should be injected, the specific injection block that should be targeted, and the bit that should be flipped. Some of the wires are arrays of multiple bits, because of this the location of the specific bit in that array is provided. All of this information is read out by the simulation and carried out during run-time.

2.4.2 Propagation checker

To gather results on whether TMR makes a difference in error propagation, a method needs to be implemented to check how many of the injected faults affect the outcome of the voter blocks. The way this is done is by first running the benchmark once without any faults injected. The outputs of the voter blocks are recorded and saved. Whenever there are errors propagated through the voter blocks, they will deviate from these expected outputs. The two results will be compared and checked for the number of differences. This will also be done using a small python script to compare the two outputs. Of course, the implementation without TMR doesn't have voter blocks thus here the direct output of the multi/div block is read and compared.

2.5 Probabilities of Single Event Effects occurring

A study from NASA has observed the amount of SEU's a satellite with a memory block encountered in orbit [10]. The SEU's were counted by using hamming error detection and correction code. Every 15 minutes the memory was read, corrected, counted and written back to the memory. The memory block could store 512Mbits and there were two present for redundancy reasons. Both memory blocks were also shielded to reduce the amount of SEU's. The average rate of Single Event Upsets was 250 per day. But because we don't want any errors propagating we shouldn't look at the average rate, but at the worst case for the number of SEU's. There are huge spikes of event upsets during solar flares. These flares release a sudden wave of charged particles which increases the number of SEU's to over 1200 in a day. This would still mean only 0.0139 SEU's per second. From this we can calculate the number of upsets per bit per second. But this would be pretty inaccurate given the big time range in which the number of SEU's were gathered. Because this was measured in orbit, the rate for events varied a lot (especially when traversing over the poles where there is a higher concentration of particles because of the magnetic field of the earth). Luckily another study by the CERN institute has been done to estimate the SEU rate [4]. This study created radiation with a flux of $4.9 * 10^7/cm^2s^1$ which resulted in an estimated SEU rate of $8.3 * 10^{-7}$ upsets per bit per second. Accounting for the number of bits that we use, which is 134 for the system without TMR, and for the time the program runs, which is 26ns it gives us an average rate of $2.89 * 10^{-9}$. Using this average rate, we can use the Poisson distribution to know the expected result(eq. (1)), where λ is the average rate and k is the number of occurrences. Using this formula, the probability that a single SEU happens in the run-time of the simulated system is $2.89 * 10^-9$, while the chance that two events happen in this run-time (Not

even at the same time or location) is $4.18 * 10^{-18}$.

$$P(SEU) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{1}$$

But, according to the NASA paper, around 20% of the telemetry files showed that multiple upsets occurred during a 10 second window. Multiple particles hitting the same memory in this time-frame is so unlikely as we've just calculated, that NASA concluded that it was most likely a single particle hitting multiple memory cells. These are called Multiple Event Upsets (MEU's) and most of these Events affected two or three memory cells. To simulate this properly the SEU rates have been drastically increased from the more realistic average values. In the appendix there are figures that show the distribution of how many faults are injected in the same cycle is given for different probabilities of a SEU happening per bit per cycle (section 5). As can be seen from the figures, starting with a probability of 0.05% chance of a SEU per bit per cycle(2ps), there are cases where 3 faults are injected at the same time. Of course the amount of faults is unrealistic but it can better demonstrate the functioning of the TMR.

3. Methods

All of the following tests are conducted using different probabilities for a Transient Event to happen. This is the probability for a Transient Event to happen on a bit, every clock cycle. Thus there could theoretically be multiple bit flips at the same time at the same place, if the probability for a Transient Event is high enough. The probabilities given are not necessarily the most realistic as discussed in 2.5, rather they are showing what TMR is capable of and what it's limits are. All the systems are running a Coremark benchmark, which utilizes the Multi/Div block. This benchmark runs for 13143 cycles. All of the results found in the tables below are averages over three runs. However the averages of 0.5% and 1.0% probability of a SET are made over five runs. This is because of the bigger fluctuation of the results. The figures of the results do include all runs, not only the averages. First, the system will be tested without any redundancy implemented. This will provide a good baseline to compare the results of the TMR with.

3.1 Without redundancy

3.1.1 Single Events Only

The first test will be conducted without TMR and with only Single Events. Because this is a simulation of a Single Event Transient there shouldn't be more than one fault injected at the same time into the system. The fault generator has been altered to make sure that if an error has been generated for a specific time, no other error can be generated in that cycle. The results can be found in table 1 and fig. 4.

Probability of SET(%)	Avg. Faults injected	Avg. Faults propagated	Ratio
0.005	87.3	5.0	0.058
0.01	190.3	15.0	0.080
0.05	845.7	52.0	0.062
0.1	1618.0	113.3	0.070
0.2	3096.3	220.7	0.071
0.5	6299.4	1576.6	0.250
1.0	9624.4	3745.4	0.389

Table 1: Average results from SET test without TMR implemented

If we look at fig. 4 we can see that the number of faults injected is not linear even though we would expect it to be. This phenomenon is most likely caused by the number of faults injected nearing the maximum number of faults possible at the higher probabilities. The maximum possible number of injected faults for this system with single events only is 13143 faults, for each cycle one. Thus a lot of faults that would normally occur are not present here which is why the number of faults injected is lower than we would normally expect. Furthermore is the ratio of number of faults injected to faults that are detected at the output very high, but maybe not as high as some would assume. The

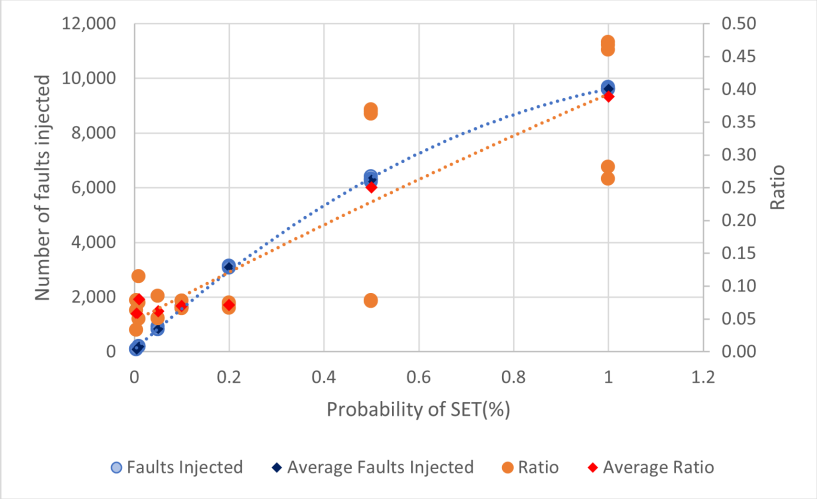


Figure 4: All results from SET test without TMR implemented

reason that not every fault injected directly results in a fault detected at the outputs of the Multi/Div block, is most likely because for a SET to be effective it must be injected when the system is actively using that specific signal. But often not all signals are used at the same time, thus the SET is injected for a clock-cycle, but it disappears afterwards. This fact also explains why the ratio goes up when there is a higher probability of bit flips. When there are more faults injected, more faults are located on signals that are being actively used. To properly simulate a particle hitting multiple wires at the same time, we need to test for Multiple Transient Events.

3.1.2 Multiple Transient Events

MET is an abbreviation for Multiple Event Transient. This means that two or more Transient Events can happen at the same time. This does not have to be the case, and is more common the more faults are injected into the system. This test just does not exclude these events anymore like the previous SET test. The distribution of these faults that will be injected can be found per probability of a SET in the appendix (section 5). The results of this test can be found in table 2 and fig. 5.

Comparing these results with the results of the SET injection, we can see that the injected faults now increase linearly. As discussed before, this is in line with the expected behaviour. Because there can be multiple faults at the same time and even in the same location, the possible number of faults injected is much higher. Again, we see the ratio increase linearly, but this time the average ratio is lower than the average ratio of SET. Especially with the higher probabilities of faults injected. When the number of faults injected is higher, the probability of faults being injected into the same location is also higher. This is likely what is happening at the higher probabilities. Two faults are injected into the same signal at the same time, this results in essentially only one fault injected which causes the ratio to go down. Now that the baseline

Probability of SET(%)	Avg. Faults injected	Avg. Faults propagated	Ratio
0.005	90.0	6.3	0.069
0.01	169.7	13.3	0.078
0.05	913.3	54.0	0.059
0.1	1731.0	115.0	0.066
0.2	3457.7	220.0	0.064
0.5	8671.6	1664.6	0.192
1.0	17338.6	5046.2	0.291

Table 2: Average results from MET test without TMR implemented

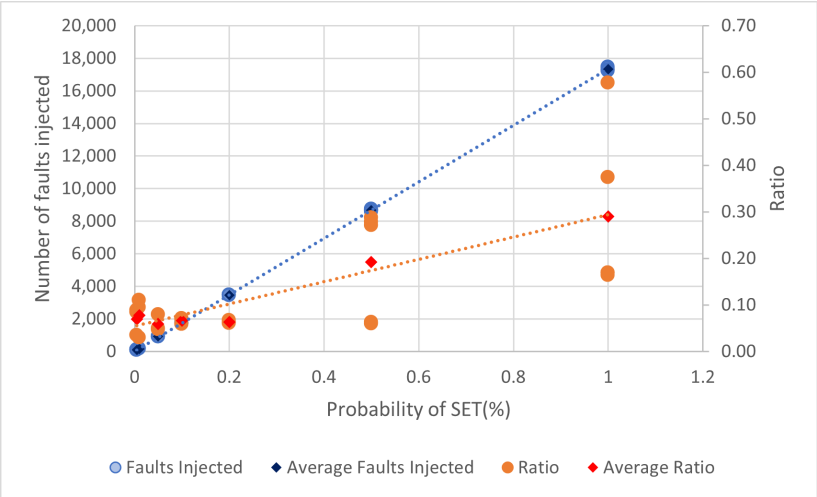


Figure 5: All results from MET test without TMR implemented

has been set, we can test the performance of the Triple Modular Redundancy.

3.2 With redundancy

This will conduct the same two test as has been done without the TMR implementation. Because of the increase in Multi/Div blocks, there are also more points where faults can happen. This means that there will automatically be more faults injected.

3.2.1 Single Events Only

Again, first the test without multiple faults at the same time is done here, the results can be found in table 3 and fig. 6.

In these results we see a big difference compared to SET without TMR. The first most obvious difference is that there are 0 faults propagated, no matter how many faults are injected. If we look at the theory of TMR this is exactly as we would expect and a different result would have indicated an incorrect implementation of TMR. TMR is always able to correct for single events. Secondly a less obvious difference is the number of faults injected.

Probability of SET(%)	Avg. Faults injected	Avg. Faults propagated	Ratio
0.005	285.0	0.0	0.0
0.01	531.3	0.0	0.0
0.05	2581.7	0.0	0.0
0.1	5187.3	0.0	0.0
0.2	10461.0	0.0	0.0
0.5	26145.0	0.0	0.0
1.0	52171.4	0.0	0.0

Table 3: Average results from SET test with TMR implemented

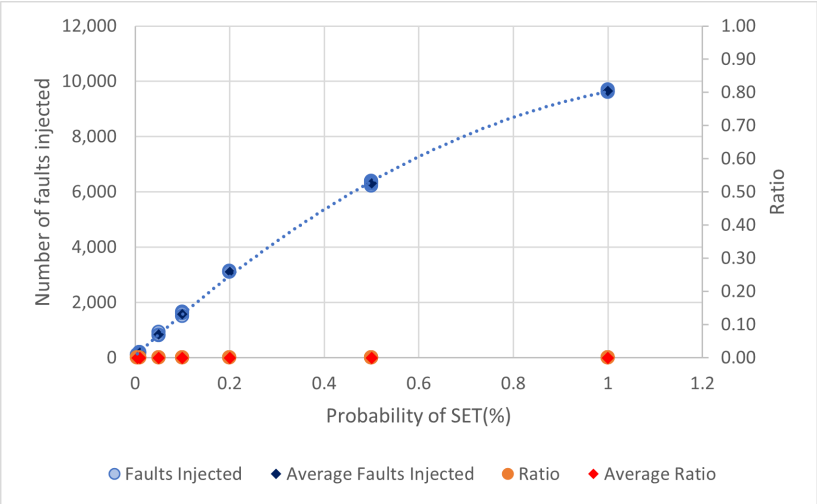


Figure 6: All results from SET test with TMR implemented

Because the hardware of the system is now increased, so is the number of wires that a Transient Event can occur. Because there are more bits that can bit flip with the same probability for each bit, there are more bit flips in total. Thirdly, the correlation between SET probability and the number of faults injected is linear which was not the case for SET without TMR. This is also because of the increase of hardware and thus the number of possible bit flip locations. To account for a particle hitting multiple wires at the same time, the Multiple Transient Events should also be simulated to test the robustness of the redundancy implementation.

3.2.2 Multiple Transient Events

The TMR will be tested were multiple faults can be injected at the same time. The results of this test can be found in table 4 and fig. 7.

This time, because the number of faults injected was not limited by the possible maximum number of faults, the number of faults injected are nearly identical of the SET case. However, here some errors are detected at the output of the voters. As can be seen from fig. 7, the ratio of faults injected versus faults propagated seems to be exponentially related. If multiple faults are injected at

Probability of SET(%)	Avg. Faults injected	Avg. Faults propagated	Ratio
0.005	262.7	0.0	0.000
0.01	514.3	0.0	0.000
0.05	2586.3	2.0	0.001
0.1	5255.7	9.7	0.002
0.2	10480.3	41.0	0.004
0.5	26245.2	194.6	0.007
1.0	52257.8	1866.8	0.036

Table 4: Average results from MET test with TMR implemented

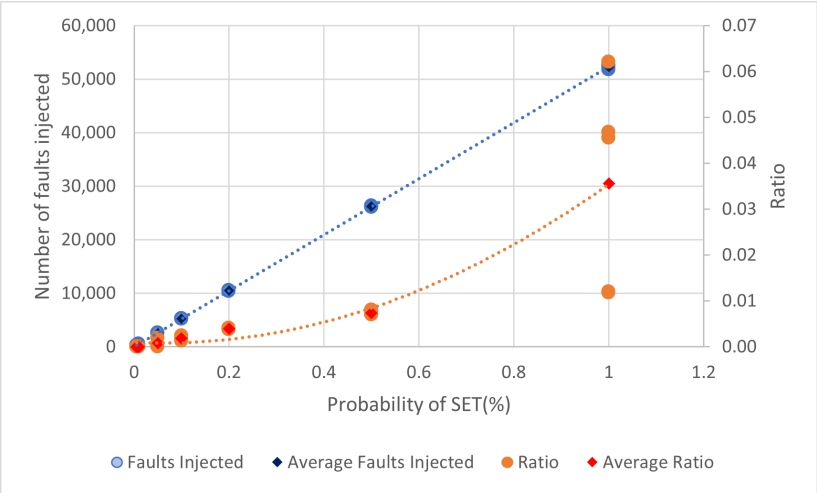


Figure 7: All results from MET test with TMR implemented

the same time at the same signal but on a different Multi/Div block, possibly an incorrect value will be propagated. Thus the more faults are injected, the higher the probability that faults of this kind are found in the system. But even if that is the case, it is very unlikely that the bit-flip will happen at the same bit position in both blocks because some signals consist of arrays of up to 34 bits. This means that there often will not be a majority at the voter block, and it will pick a random Multi/Div block input as it's output (In this system it will always pick the third output). This means that even if there are multiple bit flips on the same signal, there is still a chance that the correct output is given. Comparing the ratio of this system to the ratio of the system without TMR a huge difference can be seen. At lower SET probabilities the TMR system doesn't propagate any incorrect outputs, while at a very high SET probability the system still has a 10x lower ratio of faults injected versus faults propagated.

3.3 Transistor fault

From the previous results we observed that having TMR implemented increased the amount of faults injected because of the higher number of possible locations

where faults could occur. This raises the question, is it possible that this increase of faults could be detrimental for the reliability in certain situations? In this hypothetical situation, an transistor fault has occurred. This is a fault that can force a bit to either a zero or an one. Because a TMR system has many more components it is more likely to have such a fault somewhere in the system. To model this in the worst case scenario, a frequently used input signal of the Multi/Div block has been used and a bit in this signal has been set to 0. The reason for this is because the signal in the signal is very often a 1.

3.3.1 Transistor fault at position one

In this case, we place the stuck transistor on the first Multi/Div block. If the voter block doesn't have a straight up majority, it will pick a random Multi/Div block input as it's output. In the case of this system it is always the third Multi/Div block. Thus we expect this case to do better in comparison to a stuck transistor at location three.

Probability of SET(%)	Avg. Faults injected	Avg. Faults propagated	Ratio
0.005	255.3	2.3	0.009
0.01	523.7	3.7	0.007
0.05	2635.0	28.3	0.011
0.1	5137.7	57.7	0.011
0.2	10461.7	127.3	0.012
0.5	26140.4	296.0	0.011
1.0	52244.2	2120.2	0.041

Table 5: Average results from MET test with a transistor fault at position one

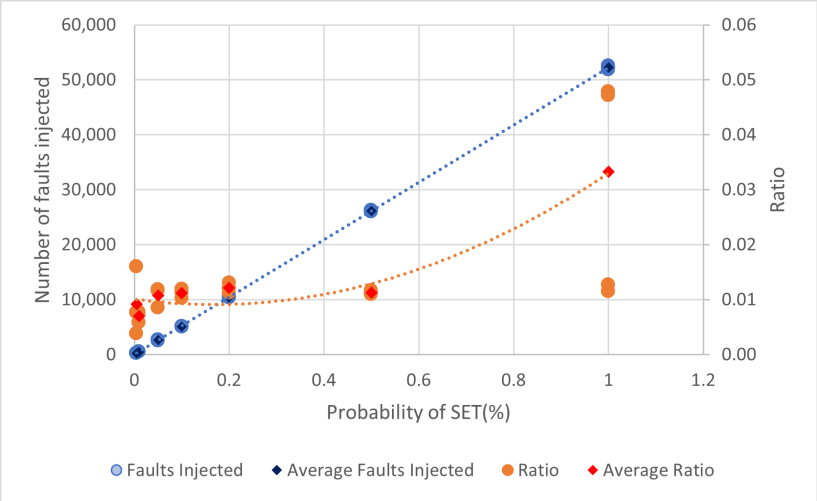


Figure 8: All results from MET test with a transistor fault at position one

The overall behaviour of the system is similar to the behavior of the regular system with TMR. One notable difference is that on lower probabilities of

a SET some faults are detected at the output of the voter blocks. This is because the threshold of a fault propagating is much lower. If an fault is injected on the signal with a transistor fault on the third position, a fault will be propagated. At higher probabilities of a SET the transistor fault isn't that noticeable anymore because it's relative impact is reduced. But even with this handicap, looking at the ratio, the system performs much better compared with a system without TMR across the board.

3.3.2 Transistor fault at position three

To now create the worst possible scenario is if there is a transistor fault at position three. This is the default input that is propagated in case of a undecided vote in the voter block. The results can be found in table 6 and fig. 9.

Probability of SET(%)	Avg. Faults injected	Avg. Faults propagated	Ratio
0.005	263,3	3,0	0,011
0.01	514,7	7,0	0,014
0.05	2634,3	28,7	0,011
0.1	5160,0	65,7	0,013
0.2	10432,7	136,0	0,013
0.5	26198,0	319,8	0,012
1.0	52529,6	1824,8	0,035

Table 6: Average results from MET test with a transistor fault at position three

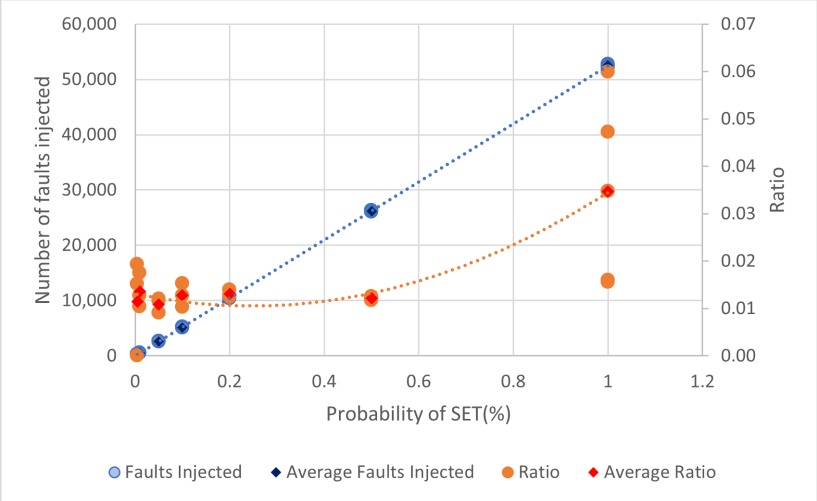


Figure 9: All results from MET test with a transistor fault at position three

As can be seen from the results, it performs slightly worse than the case with a transistor stuck at position one. Especially at lower probabilities of a SET, where the transistor fault is more significant. But even in this absolute worst case scenario, the whole system still performs much better compared to a system without TMR.

4. Discussion

When comparing the two systems, with TMR implemented and without TMR there is an obvious advantage. But as discussed before, this does come at a cost. The TMR requires extra hardware which increases cost and physical space. Therefore, TMR should only be applied to the components that need it. Even though the ratio between the injected faults versus the faults propagated is a good tool to compare the result of different SET probabilities with each other, it is not a good way to compare both systems. From an end-user perspective only the number of faults propagated matters. From the results the TMR might not seem as impressive but this is because it was pushed to it's breaking point. As calculated in section 2.5, the probability of an SET to happen even once is around $2.89 * 10^{-}9$. While we are injecting the system with tens of thousands of faults and more importantly, tens of thousands of METs. Needless to say this will never happen in real life, and the SET results are far more realistic. However, the SET results do have a minor issue that could have affected the results of the case without TMR. This is because of the way that the error generation is done. The error generator passes by every bit where a fault injection is possible, every cycle. It then decides if this bit should be flipped or not. But because the SET case could only have a single bit flip per cycle, when a fault was generated, all other bits in that cycle were skipped. Because these steps were done in the same order every time, the positions earlier in the array we much more likely to be flipped compared to the bits later in the array. This might have slightly affected the results. Future studies should be conducted using a bigger processor which consist of more flip flops that are in use. This way the effect of SEU could also be studied. The probabilities calculated in section 2.5 will also be more applicable because these are derived from probabilities of SEUs happening and not SETs.

5. Conclusion

With the emerging nanoscale CMOS technology, MET's are expected to become more frequent than SET's[3]. This means that redundancy becomes even more important in all industries. Having redundancy in your system will not automatically improve reliability. But as can be seen from the results, when redundancy is implemented in a specific way it can improve the reliability of the system greatly. Of course, the probabilities of a SET happening simulated here is much bigger than what would be normal for a regular system, but it proves the effectiveness of implementing the redundancy scheme. As discussed before, TMR will struggle when there are too many bit flips such that the voter blocks don't have a majority. But even with this being the case, at the very worst case scenario, it still outperformed having no redundancy implemented by a big margin. With having triple the amount of faults injected, it reduced the number of faults propagated by 2.7x. But in a more realistic environment, the TMR would correct for every fault injected. The system can even correct for a transistor fault where it performed only slightly worse compared to a case without a transistor fault. Such a transistor fault would be catastrophic for a system without TMR, even when there are no extra faults being injected into the system. This shows how redundancy can greatly improve reliability. But having a TMR might not be enough for your application. The NASA study discussed earlier mentioned that they had encountered charged particles that could hit up to 30 memory cells[10]. While the memory blocks they used are much denser and thus it is easier to hit multiple memory cells at once, there is a chance that if 30 memory cells are affected, TMR might not be enough. For some systems the consequences for a single error to propagate past the TMR might be too great to leave it up to chance. In this case TMR is still very valuable because as could be seen from the results it reduces the number of faults propagated drastically but it should be combined with some other form of redundancy such as information redundancy.

Bibliography

[1] Nilesh Badodekar. *Mitigating Single-Event Upsets Using Cypresss 65-nm Asynchronous SRAM.* URL: https://www.infineon.com/dgdl/Infineon-AN88889_Mitigating_Single-Event_Upsets_Using_Infineon_65-nm_Asynchronous_SRAM-ApplicationNotes-v04_00-EN.pdf?fileId=8ac78c8c7cdc391c017d07389b985c7d.

[2] K Ding. "Soft Errors Protection at Hardware, Software, and Model Levels". In: *Computer Safety, reliability, and security: 37th international conference*. SPRINGER INTERNATIONAL PU, 2018, pp. 252–252.

[3] Mojtaba Ebrahimi, Hossein Asadi, and Mehdi B. Tahoori. "A layout-based approach for Multiple Event Transient analysis". In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–6. DOI: 10.1145/2463209.2488858.

[4] F Faccio, C Detcheverry, and M Huhtinen. "Estimate of the single event upset (SEU) rate in CMS". In: (1998). URL: https://cds.cern.ch/record/405088.

[5] *FAQ full general public Webb Telescope/NASA.* URL: https://www.jwst.nasa.gov/content/about/faqs/faq.html.

[6] Steve Heath. *Embedded Systems Design (second edition)*. Newnes, 2003.

[7] Israel Koren and C. Mani Krishna. "Preliminaries". In: *Fault-Tolerant Systems* (2021), pp. 1–10. DOI: 10.1016/b978-0-12-818105-8.00011-5.

[8] lowRISC. *LowRISC/IBEX: Ibex is a small 32 bit RISC-v CPU core, previously known as Zero-riscy.* URL: https://github.com/lowRISC/ibex.

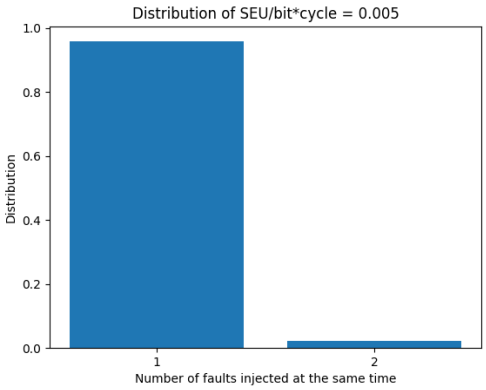
[9] S. Mitra et al. "Robust system design with built-in soft-error resilience". In: *Computer* 38.2 (2005), pp. 43–52. DOI: 10.1109/mc.2005.70.

[10] C. Poivey et al. "In-flight observations of long-term single-event effect (see) performance on OrbView-2 Solid State Recorders (SSR)". In: *2003 IEEE Radiation Effects Data Workshop* (). DOI: 10.1109/redw.2003.1281357.

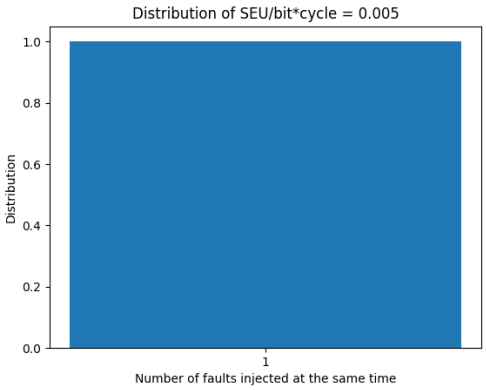
[11] Wilson Snyder. *Verilator.* URL: https://www.veripool.org/verilator/.

[12] Y.C. Yeh. "Safety critical avionics for the 777 primary flight controls system". In: *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)* (Nov. 2001). DOI: 10.1109/dasc.2001.963311.

Appendix

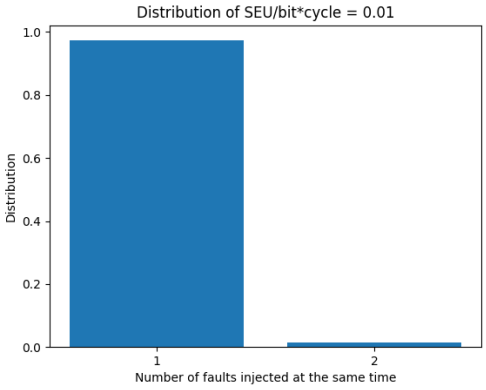


(a) With TMR

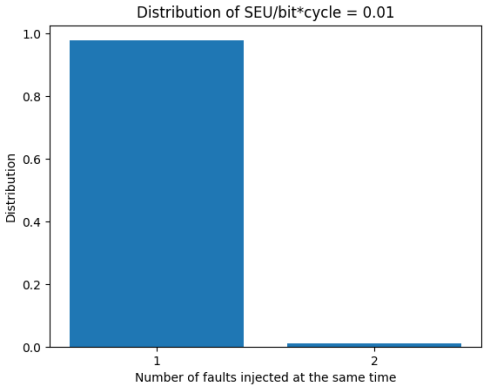


(b) Without TMR

Figure 10: The distribution of faults for a SEU/bit*cycle probability of 0.005

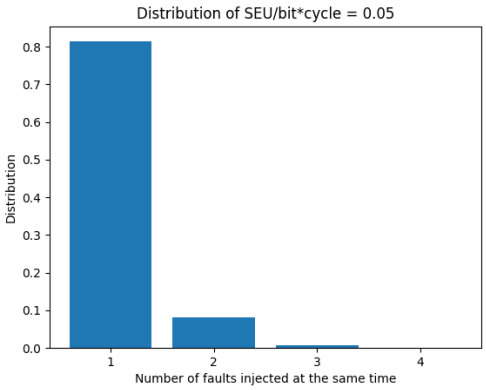


(a) With TMR

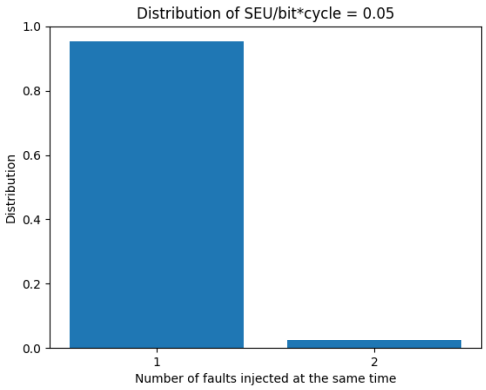


(b) Without TMR

Figure 11: The distribution of faults for a SEU/bit*cycle probability of 0.01

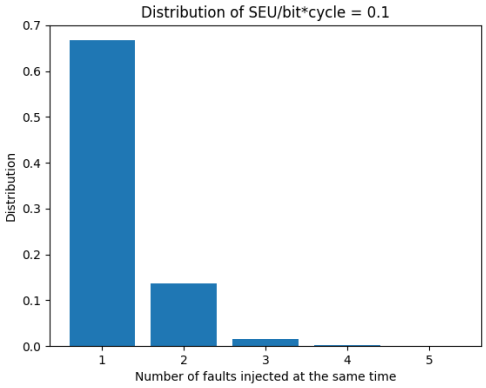


(a) With TMR

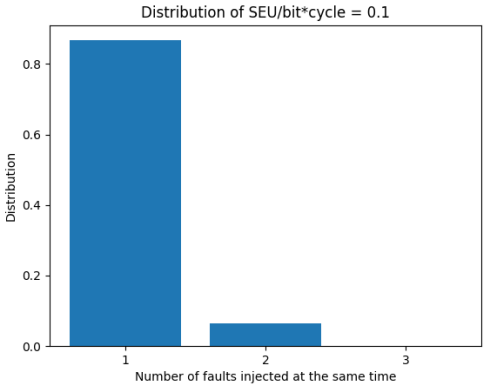


(b) Without TMR

Figure 12: The distribution of faults for a SEU/bit*cycle probability of 0.05

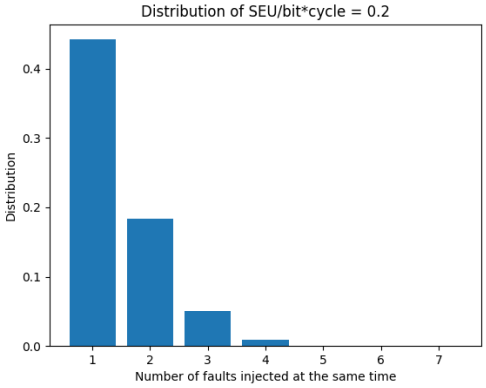


(a) With TMR

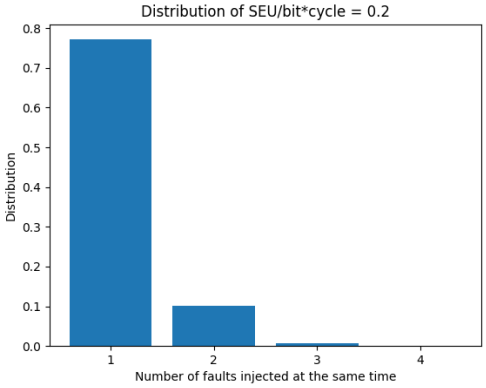


(b) Without TMR

Figure 13: The distribution of faults for a SEU/bit*cycle probability of 0.1

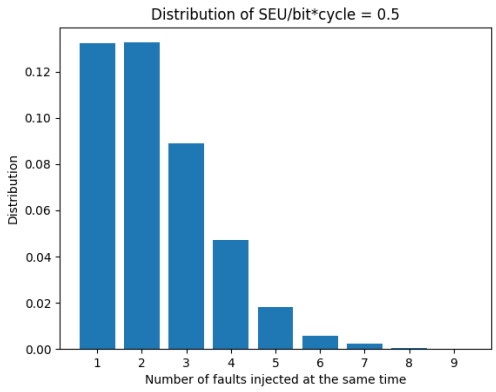


(a) With TMR

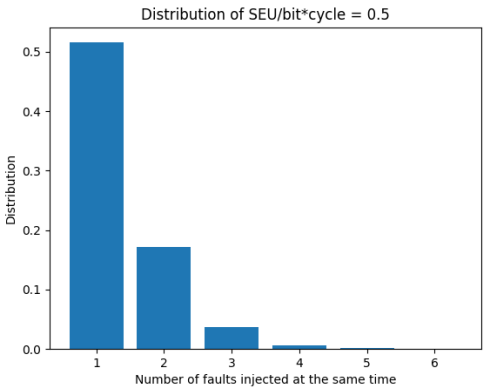


(b) Without TMR

Figure 14: The distribution of faults for a SEU/bit*cycle probability of 0.2

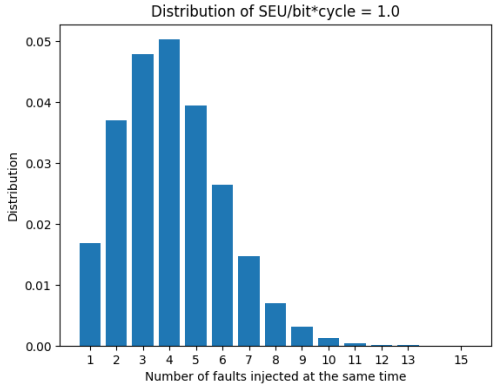


(a) With TMR

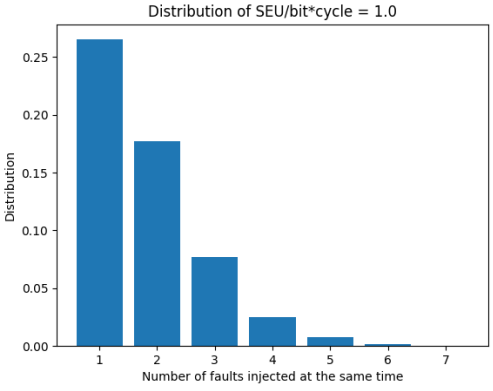


(b) Without TMR

Figure 15: The distribution of faults for a SEU/bit*cycle probability of 0.5



(a) With TMR



(b) Without TMR

Figure 16: The distribution of faults for a SEU/bit*cycle probability of 1.0