Performance Study of Different Fault Tolerance Techniques with Simple Pipelined CPU

Yuewen Lei
lei8@wisc.edu

Jing Tu
jtu2@wisc.edu

Tongxin Zheng
Tzheng9@wisc.edu

Abstract - In this project, we implemented three redundancy methods on a five-stage pipelined CPU in Verilog to realize fault tolerance, which are information redundancy, hardware redundancy and time redundancy. We ran branch intensive, arithmetic operation intensive and load store intensive benchmarks on each redundancy implementation respectively, to obtain their failure rates under different fault occurrence circumstances. We also synthesized our redundancy design to find the area overhead and maximum clock frequency. By comparing failure rate, area and performance overhead, we concluded the advantages and disadvantages of each redundancy method when it comes to their practical use.

I. INTRODUCTION

A. Problem statement

In this project, we studied and compared the performance of various fault tolerance techniques, selected from the categories of hardware redundancy, information redundancy and time redundancy. Systems with different fault tolerance techniques were implemented in Verilog based on a simple five-stage pipelined CPU. Simulation with fault injection using three benchmarks was conducted to evaluate their performance. Synthesis was also conducted to obtain some hardware statistics. Area overhead, time overhead, maximum clock frequency and correctness rate are the primary metrics we use as indicators of performance.

B. Motivation

As the design complexity increases, a computer system is more vulnerable to all kinds of malicious or accidental negative effects in which case the system could fail when an unexpected event occurs. Fault tolerance is a setup or configuration that prevents a computer or network device from failing in the event of an unexpected problem or error and thus is a necessity for high reliability, availability and security level. Generally, fault tolerance techniques can be classified into four categories, namely, hardware redundancy, information redundancy, time redundancy and software redundancy. And different categories have their distinctive advantages and disadvantages when it comes to fault tolerance efficiency. Therefore, a performance study of several fault tolerance techniques in different categories is an interesting topic to delve into. Since our focus is fault tolerance techniques, a simple platform, i.e. a simple five-stage pipelined CPU, is chosen for performance evaluation.

C. Background

Three out of the four aforementioned redundancy categories are our research subject. Hardware redundancy uses extra hardware to tolerate faults, including passive, active and hybrid redundancy. Information redundancy is achieved by using error detecting and correcting codes. Time redundancy is to accomplish a computation task for more than once over time.

Generally, a simple pipelined CPU has five stages: instruction fetch, decode, execution, memory and write-back stage. In instruction fetch stage, one instruction is fetched from instruction memory. In decode stage, the instruction is decoded to generate control signals for next three stages. Register file reading is also conducted in this stage. In execution stage, an ALU (arithmetic logic unit) is used to process data and a separate address adder is used to calculate PC address for jump and branch instruction. In memory stage, some data is written into data memory or some data is read from data memory. In writeback stage, data is written back to register file if needed. Between the stages, pipelined registers are used to store the intermittent results. Within each clock cycle, only one instruction is fetched and issued. In addition, data forwarding and bypassing unit are used to avoid data hazards.

As for fault injection, it has been found that there are mainly two types of techniques: hardware and software. Hardware-implemented fault injection uses additional hardware to introduce faults into the target system’s hardware. Software fault injection is achieved by inserting the fault injector into the application itself or layering it between the application and the operating system [1]. Because we will implement the system using Verilog, it is easier to simply add additional fault injection
hardware to the pipelined processor. Referring to [2], we have come up with a simpler design of injector which is discussed below.

For sequential circuit, faults are injected at the input of flip-flops using a mux. For combinational circuit, faults are injected at the input of combinational circuit using a mux. Injection signal is asserted if we would like to use the random data to replace the original input.

![Fault Injector Designs](image)

**Figure 1 Fault Injector Designs**

**D. Objectives**

The objectives of this project include

1) Mastering the generic methodology of studying fault tolerance techniques;

2) Forming a basic understanding of different categories’ features;

3) Quantitatively comparing the performance of different fault tolerance techniques.

The rest of the report is organized as follows. A detailed description of each fault tolerance technique is given in II together with the fault models. III discusses performance evaluation and analysis. Concluding remarks are written in IV.

**II. FAULT TOLERANCE TECHNIQUES**

**A. Hardware Redundancy**

1) Implementation

Hardware redundancy uses extra hardware to support the system being fault tolerant. Control unit and ALU are two parts we applied the hardware redundancy method to in the five stage pipelined CPU design. Control unit sets up all the control signals in the decode stage. If there is one signal having stuck-at or transient fault, this might lead to a wrong operation. ALU works on all the calculations which is the most important part in a CPU design. In this project, we triplicate control unit and ALU to achieve hardware redundancy. There are three identical modules of control unit and ALU. We take control unit as an example to explain how it works. The input signals to all the three control units come from the instruction fetch-instruction decode pipeline register. Then the signals go through all three control units to set up all the control signals. In a case that there is a fault occurring inside one control unit, the output of that control unit might be different from the other two. In order to solve this difference, a bitwise voter using majority principle is used to set up the overall output result. For example, all the three control units have the output signal Regwrite, and Regwrite_0 comes from the first control unit, Regwrite_1 comes from the second one and Regwrite_2 comes from the last one. If Regwrite_0 differs from the other two, then by taking majority principle, the voter gives us a result value equal to the value of either Regwrite_1 or Regwrite_2, which is correct.

2) Fault Model

In order to model a fault occurring inside the control unit or ALU, a number of randomly selected input bits to those two units are inversed. In this way, we do not need to inject faults inside the module, and what we do is to inject faults in the inputs which can also model faults occurring inside the module. Besides, it is much easier to develop and analyse. There is a random number generator in the design, and it generates several random numbers between 0 and 164. The reason for choosing this interval is that the total number of input signals to three control units and three ALU units is 165. The following example explains how the fault is modelled and injected. If Carryin_correct signal is the correct input to the three ALUs, and Carryin_0, Carryin_1, Carryin_2 are the actual inputs to the three ALUs and their bit locations among the interval [164, 0] are 100, 101, 102. If a random number is 100, then Carryin_0 should be the inverse of the correct input, so Carryin_0 = ~Carryin_correct. Since 101 and 102 are not selected, Carryin_1 and Carryin_2 should be equal to Carryin_correct. In this case, a fault Carryin_0 is injected into the system. In this design, up to five random numbers can be generated at the same time so we have at most five faults in the system simultaneously. The reason for not choosing more fault numbers will be discussed in the next section.

**B. Information Redundancy**

1) Implementation
For the information redundancy implementation, a single error correction double error detection (SEC-DED) code was adopted. Because information redundancy cannot be used for data transformation, we added the information redundancy hardware in the memory unit to ensure data integrity. An encoder and a decoder for error correcting code were added at the inputs and outputs of the instruction memory respectively, and the same was done for the data memory. As the information bits are 16-bit RISC instructions in the pipelined processor, we need 6-digit check bits to achieve SEC-DED according to the equation $2^r \geq n+1$ for $(n, n-r)$ code. Hence, the (22, 16) block code was used.

The generator matrix $G$ was used to encode the 16-bit information bits.

![Figure 2 G matrix of the (22,16) SEC-DED code](image)

The encoder hardware was built based on the above $G$ matrix, with six eight-input XOR trees for generating the six check bits respectively. Inputs of the XOR tree are listed in the last six columns of the $G$ matrix. XOR tree is implemented to reduce delay and glitches caused by path length difference. After encoding, the six bit checks are added to the end of the original 16-bit memory input and the new 22-bit memory entry is stored.

The parity matrix $H$ shown below is used for calculating the syndrome and correcting the faulty words.

![Figure 3 H matrix of the (22,16) SEC-DED code](image)

The decoder for the encoded word is implemented as follows. Firstly, the highest 16 bits of the 22-bit word are taken as information bits to feed into XOR tree, which is the same as the encoder structure, for generating expected check bits. Then the expected check bits are compared with the actual check bits by XORing them bit-wise to create a syndrome for this data. Syndrome is then fed to the syndrome decoder so that the position of the one-bit fault can be translated. The syndrome decoder is constructed according to the above $H$ matrix. When a syndrome coincides with a column of the $H$ matrix, the fault is diagnosed to have taken place on that corresponding bit. If the syndrome does not match any column of the $H$ matrix, it can be deducted that more than one bit of the word are mistaken. If the syndrome is all zero, meaning that there is no fault, the syndrome decoder is disabled. Finally, the fault is added to the input word for correction. If two or more bits are erroneous in the word, then a double error signal is set to inform the process that an uncorrectable error occurs.

For the reason that encoded words are of length 22 bits, we need to augment memory to hold the extra six bits. In addition, the data or benchmark stored in instruction memory should be converted to be 22 bit encoded word. A script was written to perform the encoding and conversion.

2) Fault Model

To evaluate the performance of the SEC-DED code hardware, faults should be injected. According to the fault injection method in [2], we need to insert fault insertion hardware randomly in the memory unit. Because this will cost much extra hardware and is repetitive and cumbersome, an equivalent way of fault injection came up. Because not all memory entries are fetched in the benchmarks, faults inserted inside the memory may never get excited and propagated. Hence, instead of adding fault-inserting multiplexers in the memory, we placed 22 2-to-1 multiplexers at the output of the memory module, one for each bit of the memory output. Each MUX has one input connected to the faulty value and the other input connected to the correct data bit. By setting selection signal of the MUX to 1, fault is inserted. The selection signal is generated randomly so that the fault randomly takes place in different bits of the output.

C. Time Redundancy

1) Implementation

Time redundancy is, as its name indicates, to use more time to tolerate faults, which is always achieved through re-execution. In this project, we only execute those instructions that need ALU multiple times since our ISA is RISC and ALU is the most error-prone. Most of the modifications to the baseline processor are in the execution stage. A finite state machine that controls the re-execution behavior and a correction unit that is used to
correct the faults are added to this stage. An extra control signal for the execution stage is generated from the decode stage to indicate whether the instruction passed to the execution needs the ALU operation. Since our baseline processor is pipelined, during the re-execution, all pipeline registers together with the PC register are stalled. Thus, five control signals are generated from the finite state machine. The main functions of the correction unit is to detect the occurrence of a fault and correct it. To detect a fault, the result of the first time execution is stored in a register and a second execution of the instruction is conducted with the result compared to the previously stored one. If they are not equal, a fault is supposed to be detected and a signal called faulty is asserted to notify the finite state machine that further re-execution is needed to correct the detected fault. To correct the fault, the result of the second execution is stored in another register and a third execution of the instruction is needed. A majority voter is then used to restore the correct execution result among the results through these three times' executions.

Since the only circumstance when the voter can produce the correct result is when two among the three execution results are correct, which means this simplex design can only deal with transient faults that last less than one clock cycle and is useless with the occurrence of permanent faults that persist after its occurrence. Hence, an improved design is implemented to target transient faults and permanent faults through RESO (re-execution with shifted operands) proposed in [4]. Two shifters are placed in front of the operand ports of the ALU together with two multiplexers to choose between the original value and the shifted value. In the correction unit, shifters are added to shift the execution result back when needed. The reason why RESO would work for permanent fault is that through shifting operands and results around during re-executions, the bit line fault occurs are different which guarantees that for any bit position among the three results, we always have two correct values that contribute to the majority voting. Similar to the explanation of the simplex design, we now go over the fault detection and correction process.

When the finite state machine observes an asserted need ALU signal, it stalls the pipeline registers and the PC register and tells the correction unit to store the result of this execution. When the next clock cycle comes, it generates signals to control the shifters in front of the ALU operand ports to shift the input left by 1 bit to produce the shifted operands and the two multiplexers also select the shifted operands for ALU operation. When the result of this re-execution enters the correction unit, a shifter is used to shift the result right by 1 bit to restore the ALU result. The restored result is compared with the previously stored one to find whether a fault occurs. If they are equal which means no fault, the finite state machine lets the pipeline registers and PC registers move on to process next instruction. Otherwise, those registers will be stalled for another clock cycle for the third time's execution and the restored result is stored in another register for future voting. During the third time's execution, the shifters in front of the ALU operand ports shift their input left by 2 bits and again the multiplexers select the shifted operands. When the newly generated result enters the correction unit, another shifter shifts the result right by 2 bits and this restored result together with the two stored execution results enter the voter to generate the corrected result. To guarantee the ALU precision with the 16-bit operands shifted around, the original 16-bit ALU is extended to 18-bit one. A special case needs to be taken care of when a jump or a taken branch comes to the picture. In our baseline design, the PC change of these instruction happens after the execution stage when the next instruction of the jump or branch, which is supposed to be flushed, already enters the execution stage. If this instruction needs ALU operation, it will trigger the finite state machine to stall the PC register which is not right. Hence, the finite state machine needs feedback from the EXE_MEM pipeline register to tell whether the last instruction is a taken branch or a jump. If so, the re-execution behaviour is not triggered.

2) Fault Model
To evaluate the fault tolerance performance of our design, we need to model the faults in the ALU. Since our design is based on hardware description language, i.e. Verilog, the injection of faults into the interior design of the ALU would be effort-consuming and cumbersome. So we take an alternative by injecting faults on the ALU output. Here is some justification. When some faults occur inside the ALU circuit, they have to be propagated to the output to cause potential performance deviation. Otherwise, the faults don’t need to be detected and corrected. When the faults manifest themselves on the output line, they exhibit randomness on the bit positions because different faults have different effect range on output bit lines. Besides, the manifestation of the fault should differ from the original correct value for its detection and correction. Otherwise, the faults still don’t have performance influence. Based on these arguments, we came up with the fault injector and fault generator to model the fault behaviour. The fault injector simply Exclusive-OR the correct ALU results with a fault vector. In the fault vector, a ‘0’ in one bit position means there is no error in the corresponding output bit line while a ‘1’ indicates the occurrence of an
error in the bit line. And the fault generator is responsible for generating this fault vector. The number of ‘1’s in the fault vector determines the number of errors in the ALU output and we allow up to 4 simultaneous errors. And for the simplex time redundancy design, we only model transient faults since the design lacks the capability of dealing with permanent faults.

III. PERFORMANCE EVALUATION

A. Synthesis Results

To evaluate the performance of our designs, we ran synthesis to obtain the area overhead and maximum clock frequency. We use Synopsys DC Compiler to perform the synthesis and the library we use is 45nm gate library provided by FreePDK. Table I lists out the synthesis results of each design focusing on area and maximum clock frequency.

<table>
<thead>
<tr>
<th></th>
<th>Total Cell Area</th>
<th>area ratio (compare to original)</th>
<th>Maximum clock frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>53549.47547</td>
<td>1</td>
<td>300MHz</td>
</tr>
<tr>
<td>Hardware redundancy</td>
<td>67604.54086</td>
<td>1.262</td>
<td>300MHz</td>
</tr>
<tr>
<td>Information redundancy</td>
<td>76324.85373</td>
<td>1.425</td>
<td>300MHz</td>
</tr>
<tr>
<td>Time redundancy (RESO)</td>
<td>58211.03226</td>
<td>1.09</td>
<td>250MHz</td>
</tr>
<tr>
<td>Time redundancy (Simplex)</td>
<td>54790.77394</td>
<td>1.02</td>
<td>300MHz</td>
</tr>
</tbody>
</table>

In hardware redundancy implementation, additional two control units and two ALUs are added to the system which will definitely cause the area to increase largely. From the table above, the area for hardware redundancy is 67604.540864 which is about 1.262 times the original five stage pipelined CPU. The additional area is not like our expected large, and our explanation is that the system has two big memory units, so the increased area is not too large compared to the original area. We added a lot of bitwise voter to the output of control units and ALUs and if the original critical path is in control units or ALUs, the maximum clock frequency would decrease. However, after the synthesis, we found the critical path is not in those two regions, and that explains why the maximum frequency does not change in hardware redundancy implementation.

In information redundancy implementation, we can see that the total area is 76324.60 which is about 1.4 times the original circuit. Of all the hardware overhead, more than half is contributed by the extra memory used for storing the check bits. The maximum frequency is not affected by adding the encoder and decoder at the memory unit because the critical path is not in the memory write cycle.

In time redundancy, we can see that the simplex design hardware overhead is negligible since extra time is paid to achieve the fault tolerance. And the RESO design has 9% overhead due to the addition of shifters and the extension of ALU. Besides, the maximum clock frequency of RESO design is smaller than the baseline because of the prolongation of the critical path.

B. Simulation Results

1) Hardware Redundancy

With the fault model mentioned above, a failure rate analysis with respect to the number of faults in the system can be developed. In our simulation, a fault free version of design with three benchmarks (branch intensive, arithmetic operation intensive and load store intensive) is first simulated. Then we inject one fault to the system and simulate 10 times for each benchmark with different fault locations with the help of the random number generator. We run three benchmarks because in some benchmarks, the fault may not manifest. We probably should run more times with different fault locations instead of just 10 times so we can have a more accurate result. By comparing with the correct result, we can know whether this time the fault manifests and causes a failure in the system or not, then by repeating 9 more times, a failure rate for one fault is achieved. We inject more faults to the system to get the relationship between failure rate and fault number in hardware redundancy implementation.

The simulation result is shown in Figure 4 and Figure 5. In Figure 4, we can see that the hardware redundancy can tolerate 1 fault. But with more faults in the system, different benchmark has different failure rate, which means in some benchmarks, a fault may not
manifest, and so there is no failure in the system. If we only focus on one specific benchmark’s failure rate at a time, we can see that load store intensive benchmark’s failure rate increases with the increasing fault number. For branch intensive benchmark, the failure rates are the same when fault number is 3 and 4. However, for arithmetic operation intensive benchmark, the failure rate decreases when fault number is 4. Our explanation to this situation is that we only did 10 times experiments for each benchmark at a fixed fault number, some randomly selected fault locations may not cause a failure in some specific benchmarks. If we do more experiments, we should have a better graph. We stopped our experiment when the fault number is five, which is because at fault number five, we have a very high probability that the system will fail, so it will be meaningless if we continue to increase the fault number and do the experiment. In Figure 5, we combined three benchmark results to form an average failure rate according to fault number. It is obvious that the failure rate increases with the increasing fault number. The hardware redundancy works very well. If the fault number is not larger than or equal to three.

![Figure 4 Failure Rate as Fault Number Increases for Each Benchmark](image)

![Figure 5 Average Failure Rate as Fault Number Increases](image)

2) Information Redundancy

In this project, we simulated stuck-at-1 fault and stuck-at-0 faults for both instruction and data memory by applying 1 and 0 to the faulty input of MUX respectively. Simulation was done by writing a testbench to asserting different number of multiplexers’ selection signals at different clock cycle. When only one MUX’s selection signal is set to 1, the fault is guaranteed to be corrected. When two and more bits are selected to be faulty, it is expected to see a double error signal. As for the benchmark of the processor instructions, we used the three programs mentioned in hardware redundancy part. Results for the simulation are discussed below.

When inserting faults in the instruction memory, if there is only one bit error, the results are always correct. If there are more than two errors, the outputs are faulty because the instructions are completely wrong and we could see the double error signal set high.

When inserting faults in the data memory, the correctness of the processing result depends on the benchmarks. For arithmetic intensive and branch intensive benchmarks, when there is few accesses to the memory, inserting more than two faults at the memory output will not lead to wrong computation results especially for simulating stuck-at-0 fault. This is because the data inside memory are initialized to zero and little of the memory data is accessed or modified. For load-store test where memory is frequently read or write, the necessity of error correcting code is more obvious. Although for stuck-at-0 faults, the correctness is not much influenced due to different benchmarks. When there are more than 2 bit errors for simulating stuck-at-1 fault, the result is wrong and the processor sets the double error
signal. In general, disregarding the impact of benchmark, we can guarantee the system to be 1 digit per instruction word correctable. This is especially useful for ensuring instruction memory to be correct.

3) Time Redundancy

![Failure Rate vs Error Number](image)

Figure 6 Failure Rate as Fault Number Increases for Each Benchmark

For the simplex design, we only inject transient faults since it is incapable of detecting and correcting permanent faults. And the simplex design can handle as many transient faults as possible at a time since the faults only last for one clock cycle and the voter always functions correctly.

As you can see from figure above, when there is only one erroneous bit line, the error is totally detectable and correctable. When the error number becomes larger than one, the system has certain possibility to fail. As the error number increases, the possibility of system failure increases linearly. When the error number reaches 4, the system could barely work. The explanation for this performance degradation is from our design of how the shifters are used and the working condition of the majority voter. Since the operands are shifted to the left by one bit and two bits during the second and third execution, after the ALU result is shifted back in the correction unit, the error bit line is also shifted by one bit and two bits. Hence, when the multiple error positions are close, there is still some possibility that one shifted erroneous position overlaps with another erroneous position which leads to a voting failure. For example, we assume bit 3, bit 4, and bit 5 of the ALU output are erroneous. For the restored result of the second execution, bit 2, bit 3 and bit 4 are erroneous. And for the restored result of the third execution, bit 1, bit 2, bit 3 are erroneous. So after the voting, bit 3, bit 2 and bit 4 are still erroneous. Generally, when the distance between any two error positions is larger than or equal to 3, the RESO design will work. Otherwise, the system will fail. So as the error number increases, the possibility of any two error positions being within 2 bit away from each other becomes larger, which explains the simulation results.

Below is a table recording the number of clock cycles used to execute each benchmark in each processor design. As you can see, the time overhead increases as the percentage of arithmetic instructions increases. Even with no fault occurrence, the cycle count of RESO design is much larger than the baseline due to the second execution of arithmetic instructions.

<table>
<thead>
<tr>
<th></th>
<th>baseline</th>
<th>RESO(no fault)</th>
<th>RESO(one or more faults)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>51</td>
<td>80</td>
<td>109</td>
</tr>
<tr>
<td>loadstore</td>
<td>41</td>
<td>69</td>
<td>97</td>
</tr>
<tr>
<td>branch</td>
<td>66</td>
<td>75</td>
<td>84</td>
</tr>
</tbody>
</table>

IV. CONCLUSION

In this project, we implemented hardware redundancy, information redundancy, and time redundancy technique on a five-stage pipelined CPU and injected faults into each redundancy implementation respectively. Both simulation and synthesis are performed to get the failure rate, area overhead and maximum clock frequency. Using hardware redundancy, the area is 1.262 times larger than the original design, but the maximum clock frequency is not changed. The failure rate simulation shows that if the fault number is less than or equal to 2, the hardware redundancy performs very well. But if the fault number is larger than 2, the system is much easier to fail. In information redundancy, the hardware overhead is the largest out of the three redundancy schemes, but the fault tolerance is also highest. With the error correcting code, the memory unit is able to tolerant up to 1 bit error per memory entry and is able to detect two or more errors. In time redundancy, the simplex design has the smallest area overhead but has limited fault detecting and correcting capability. The RESO design has larger area overhead but the overhead is still smaller than the other two redundancy design. Also, the maximum
clock frequency suffers 16% degradation. The sacrifice is traded for the detection and correction of permanent faults. And the simulation results show that the system is likely to fail with more than two simultaneous fault occurrences. Furthermore, the execution cycle count increases with time redundancy and the increasing degree depends on the arithmetic instruction percentage. Table III lists the comparison results among these three implementations.

In summary, information redundancy is crucial to guarantee the data integrity which is orthogonal to the other redundancy techniques. And the extra storage is necessary even though it’s a large overhead. As for hardware redundancy and time redundancy, they are the two extremes of the design space. On one hand, time redundancy exerts plenty of pressure on timing which could lead to clock frequency degradation and longer execution cycles. But the area overhead is little. Hardware redundancy, on the other hand, consumes extra space to reduce the possible performance degradation.

<table>
<thead>
<tr>
<th></th>
<th>Implementation Efforts</th>
<th>Fault-tolerance</th>
<th>Area overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Redundancy</td>
<td>Low</td>
<td>Only tolerate one fault; With fewer than 3 simultaneous faults, the system could work depending on the fault manifestation.</td>
<td>Medium</td>
</tr>
<tr>
<td>Information Redundancy</td>
<td>Medium</td>
<td>Only tolerate one fault per data line; System fails when multiple faults occur.</td>
<td>High</td>
</tr>
<tr>
<td>Time Redundancy</td>
<td>Medium to High</td>
<td>Simplex design only tolerates transient faults; RESO tolerates one fault; When multiple faults occur, the erroneous positions determine whether the system fails.</td>
<td>Low</td>
</tr>
</tbody>
</table>

REFERENCES