Microprocessor Sensitivity to Failures: Control vs. Execution and Combinational vs. Sequential Logic

Giacinto Paolo Saggese, Anoop Vetteth, Zbigniew Kalbarczyk, Ravishankar Iyer
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Email: {saggese, vetteth, kalbar, iyer}@crhc.uiuc.edu
1308 W. Main Street, Urbana, IL 61801, USA

Abstract*: The goal of this study is to characterize the impact of soft errors on embedded processors. We focus on control versus speculation logic on one hand, and combinational versus sequential logic on the other. The target system is a gate-level implementation of a DLX-like processor. The synthesized design is simulated, and transients are injected to stress the processor while it is executing selected applications. Analysis of the collected data shows that fault sensitivity of the combinational logic (4.2% for a fault duration of one clock cycle) is not negligible, even though it is smaller than the fault sensitivity of flip-flops (10.4%). Detailed study of the error impact, measured at the application level, reveals that errors in speculation and control blocks collectively contribute to about 34% of crashes, 34% of fail-silent violations and 69% of application incomplete executions. These figures indicate the increasing need for processor-level detection techniques over generic methods, such as ECC and parity, to prevent such errors from propagating beyond the processor boundaries.*

1 Introduction

The issue of soft errors (or single event upsets) is one of the major concerns in designing and implementing the current generation of highly integrated digital systems. Many sources [1] contribute to soft-errors, including ionizing radiation particles, capacitive coupling, electromagnetic interference, and other causes of electrical noise. Recent studies indicate that soft error rates (SER) in the data-path and combinational logic of processors are increasing [2], [3]. Even if the per-bit SER remains constant with advances in technology [1], [4], the SER per chip is expected to increase quadratically, due to increase in the number of transistors per die. Several factors, such as device and voltage scaling, increasing frequencies and pipeline lengths [1], [2] can contribute to an increase in SERs with new generations of manufacturing technologies.

This paper studies the impact of soft errors in a microprocessor intended for embedded applications. The target system is a gate-level implementation (i.e., fully synthesized design) of a DLX-like processor. The synthesized design is simulated, and transient faults (targeting combinational logic and flip-flops) are injected to stress the processor while it is executing selected applications. Collected data are analyzed to assess (i) the fault sensitivity of control and speculation logic compared to that of other functional blocks, (ii) the error contribution from combinational circuits versus flip-flops, and (iii) error propagation between the functional units. We target embedded processors because they are pervasive. Major contributions of this work are as follows:

A unique fault injection study of the impact of faults in the processor's combinational logic and flip-flops. While most of the previous studies focused on the impact of errors in flip-flops (e.g., [5], [6]), few previous studies have addressed the issue of soft errors in combinational logic [2].

Characterization of the impact of faults in control logic and speculation logic. For the analyzed workloads, the fault sensitivity of these two macro-blocks of the processor is about two (control) and five (speculation) times higher than those of the processor blocks responsible for instruction execution. While in the past focus has been on protection mechanisms for ALU [9], caches, and application-level control flow [8], the results presented in this paper indicate that in future-generation processors, control and speculation logic may become a significant source of errors that can lead to application failures. One can expect that this problem would worsen for the speculation and control logic of more complex microprocessors (such as Intel Pentium 4 and AMD Athlon 64).

Estimation of fault sensitivity of processor components. The analysis highlights the fact that fault sensitivity of different functional units ranges from 3% for the *Reorder buffer* to 44% for the *Bus interface unit*. Moreover, the *Register file*, *Bus interface unit,* and *Load-store unit* contribute to more than 50% of the errors manifested outside the processor boundaries.

Characterization of the impact of processor errors on application. Our analysis indicates that *Register file*, *Instruction fetch*, and *Dispatcher* account for 70% of application crashes. Even though the *Bus interface unit* has high fault sensitivity, there is a 34% probability of the fault being subsequently masked by the application. The data also show that a fault in the *Commit unit* is more likely to cause an incomplete execution (roughly 3 times more frequent) than any other functional unit.

Analysis of fault propagation between the functional units. Twenty most frequent fault propagation paths account for about 50% of all traversed paths. This information can be used to deploy protection mechanisms to maximize the coverage with low area overhead.

2 Target System

The target system is a superscalar version of the DLX RISC processor [11]. We started from a publicly available mixed behavioral/RTL VHDL model description of a superscalar DLX [12]. The VHDL design of the DLX processor was modified in order to make it synthesizable. The design hierarchy is extracted from the original description in order to selectively inject faults in various portions of the processor and

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

to trace the effect of the faults across the functional units. To enable the application execution (compiled C programs) on the simulated processor and collection of the experimental results, the design is augmented with additional behavioral blocks including: (i) memory to load/store data and instructions and (ii) a test bench to load data from a file to the memory and to dump the content of the memory to a file. A brief description of the target processor and its main blocks is given in the following sections.

2.1 Processor Components

The processor is divided into four functional blocks: (i) *execution*, (ii) *control*, (iii) *speculation*, and (iv) *memory interface*. Individual blocks are further divided into several functional units as depicted in Figure 1.

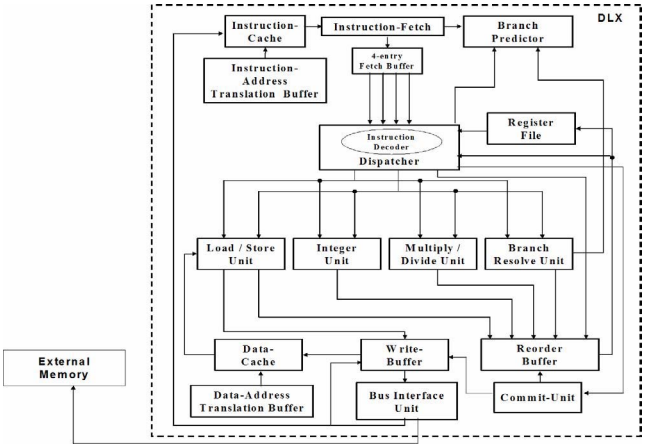

Figure 1: Architecture of the target (DLX-like) processor

Execution block

Arithmetic-logic unit (Alu) and *Multiply-divide unit (Mdu):* Form the functional units of the microprocessor responsible for executing the integer arithmetic operations.

*Register file (Rf)***:** Holds the runtime processor's architectural state, i.e., the user-level registers; also includes 64-byte instruction cache, 64-byte data cache, instruction-address-translation-buffer (4 entries) and data address translation buffer (4 entries).

Control block

Dispatcher (Dp): Decodes instructions, computes branch and jump destinations, evaluates data (such as branch conditions and jump register) for execution units, issues instructions, enables the *Instruction fetch* unit to clear and overwrite instructions, does exception handling and controls the data-path of the microprocessor.

Reorder buffer (Rb): Controls the committing of instructions in the program order. *Reorder buffer* is linked to the reservation station and generates the commit signal for the various entries in the reservation station.

Speculation block

Instruction fetch (If): Fetches instructions (up to two instructions at a time) from the instruction cache and appropriately, increments the program counter. *Instruction fetch* hosts the branch target buffer*,* which performs the branch prediction.

Branch resolve unit (Bru): Compares the branch prediction result and the computed branch condition, and flags an exception if there is a mismatch. Upon an exception, the dispatcher flushes the pipeline and issues the resolved branch condition.

Commit unit (Cu): Commits up to two instructions per clock cycle. Committing an instruction means to complete the instruction processing within the pipeline, recover all resources used by that instruction, and write-back the result into the register file.

Memory interface block

Load-store unit (Lsu): Executes the load and store instructions.

Write buffer (Wb): Maintains a queue of data to be stored in memory and performs the memory write.

Bus interface unit (Biu): Controls usage of the data bus, e.g., acknowledges the transferred data, and implements the protocol to communicate with the external memory.

Note that only *Instruction fetch* has functionalities belonging to both control and speculation blocks. The control part of *Instruction fetch* comprises only the program counter, while the major area is occupied by the branch target buffer, which does the branch speculation. Hence, *Instruction fetch* is classified as speculation block.

2.2 Processor Implementation

The target Superscalar DLX processor described in VHDL (Aldec Active VHDL 4.2) was synthesized using *Synplicity Synplify Pro 7.1* toolset integrated with the Xilinx ISE 4.1 design flow, for a Xilinx Virtex FPGA. The Xilinx Virtex series of FPGAs consists of logic cells, *Configurable logic blocks (CLBs)*, and interconnection circuitry, tiled to form a chip. Each CLB consists of two slices, and each slice as shown in Figure 2 contains (i) two 4-input *look-up tables (LUTs)* statically programmed during the bootstrap with the configuration bit-stream*,* (ii) two *flip-flops (FFs),* storage elements in the user visible system state, and (iii) associated logic, which represents fixed gates that cannot be programmed, e.g., multiplexers, selector logic, xors, and buffers.

Functional Block	Included Functional Units	LUTs		Gates		FFs		Total
		Number	% of LUTs	Number	% of Gates	Number	% of FFs	% of Circuit Element
Execution	Alu, Mdu, Rf	9020	57.5	16819	67.8	1282	47.8	62.8
Control	Dp, Rb	3817	24.3	4193	16.9	398	14.8	19.5
Speculation	Bru, Cu, If	706	4.5	917	3.7	380	14.2	4.6
Memory interface	Biu, Lsu, Wb	2142	13.7	2880	11.6	621	23.2	13.1

Table 1: Area occupation per functional block in the synthesized DLX processor

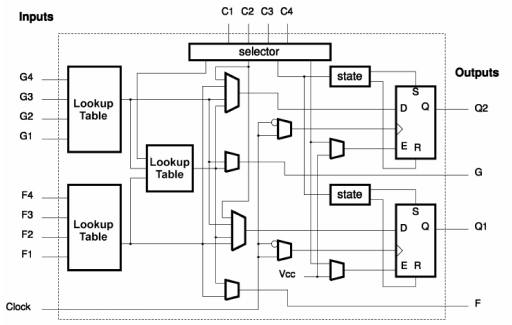

Figure 2: Basic architecture of Xilinx FPGA slice

The obtained implementation of the Superscalar DLX processor occupies 4873 flip-flops (FFs), and 16395 look-up tables (LUTs), mapped into 9526 slices (49% of the overall number of available slices in a VirtexE2000-8). The minimum clock period of the synthesized system is about 60 ns.

Table 1 reports the area occupation (in terms of structural components of FPGA) with respect to the functional blocks (execution, control, speculation, and memory interface). Observe that the area occupation of the control and speculation blocks together is about one third of the execution block.

All VHDL simulations were done using the commercial HDL simulator *Modeltech Modelsim SE 5.8c.*

2.3 Workload

Two applications (the *Bubble Sort* algorithm and *Prime Number* generator) are used as workloads to generate processor activity and to exercise the injected faults. Table 2 provides the characteristics of the two target programs. Bubble Sort is a memory intensive application, whereas the Prime Number generator is CPU-intensive. The simulation of 18µs of DLX processor activity with a clock period of 60ns requires 30 seconds on an Intel 3.0 GHz Pentium 4 processor equipped with 1 GB RAM.

Application	Number of Instructions	Simulated Execution Time
Bubble Sort	289	18µs
Prime Number	760	46µs

Table 2: Application characterization

3 Fault model

Fault model employed in this study mimics soft errors in combinational circuit and flip-flops, due to events such as a radiation particle strike. When a particle strikes a sensitive region of a transistor in a logic gate, it produces an excess charge, creating a current pulse with a rapid rise time, but a more gradual fall time (e.g., see [13]). This current pulse results in a voltage pulse, which can upset the output value of the gate, resulting in a fault. The effect of this on CMOS circuits is a voltage transient since the gate forces the output value to the stable value after the charge coming from the strike has dissipated. The duration of the fault depends on multiple factors, e.g., the charge collected due to the particle strike (which depends on the layout of the gate and the energy of the particle), the recombination time of this excess charge (which depends on the substrate characteristics), the rise and the fall time of the pulse, and the driving strength of the gate [1], [2], [14]. The distribution of the soft error duration for a given technology and cell layout can be obtained by using Spice [15] simulations, e.g., [1], [2], [16], [17]. In this study, the system behavior is analyzed for several values of fault durations, ranging from 1/8 to 1 clock cycle.

The fault model is discussed in the context of Xilinx FPGA based hardware, which is used to implement the target processor. The following hypothesis for the fault model is assumed: (i) The probability of concurrent particle strikes is negligible, i.e., a single fault model is adopted. (ii) Faults are equally distributed spatially over the FPGA components (LUTs, Gates, and FFs). (iii) Faults are equally distributed in time, i.e., uniformly distributed during the simulation time. (iv) Faults in combinational logic are transient, while faults in the flip-flops are bit-flips. (v) Faults can only occur as a functional upset in elements of user logic, but not as configuration upset (we assume that soft errors do not affect the post-power-up configuration LUT characteristics) [18].

Fault locations. A soft error can occur in different locations/points of the FPGA building blocks: *(A)* at the output of a gate, *(B)* at the output of an LUT, *(C)* at the output of a flip-flop and *(D)* in the state of a flip-flop. A fault of types *(A)* and *(B)* models the situation when a particle hits a particular transistor of a gate or an LUT, resulting in a voltage pulse at the component output causing a flip in the logic value [2]. Similarly the voltage pulse at the output of the flip-flop (type *(C)*) models the particle strike at the output stage of the flip-flop. Case *(D)* represents the strike of a particle, flipping the state of a flip-flop [1]. Fault categories *(A), (B)*, and *(C)* are transient, i.e., the steady stable state of the signal will be eventually reached because the implementing circuits are static CMOS rather than dynamic logic. The fault category (*D)* persists until the content of the flip-flop is overwritten.

Implemented fault model. The injection mechanism is generic and simulator-independent, i.e., the support for fault injection is transparent to the *Modelsim* core functionalities. *Injection of fault categories (A), (B), and (C):* To inject a transient fault with duration $[T_0, T_1]$, the output signal driven by a *Gate*, a *LUT*, or a *FF* is forced to the inverted value of the signal at the time T_0 and kept for the duration of the fault interval $[T_0, T_1]$. (This behavior is dictated by the physics of the excess charge). *Injection of fault category (D):* The output port of the flip-flop is forced to the inverted value of the flip-flop state until this state is overwritten.

Outcome	Definition
Crash	A memory location loaded or stored is out of the boundaries of the application image.
Fail silent data violation	The application terminates without crashing but the memory image is different from the golden run.
Incomplete execution	The program does not complete in the expected time (normal execution time + 10% extra time margin).
No effect	There is a mismatch at the pin-level in the cycle-accurate behavior of the processor, and the application program terminates correctly.

Table 3: Outcome categories

Outcome categories. A fault is considered as an error when it manifests at the interface of the processor, changing (com-

pared with the golden run, i.e., a run of the system in the absence of failures) the processor behavior for at least one clock cycle. Impact of these errors at the processor boundaries is further analyzed at application levels. Table 3 gives the failure categories considered at the application level.

4 Fault Injector Framework

A simulation-based fault injection framework was developed to facilitate experiments. The key features of the framework include (i) automation of the injection experiments (using gate-level circuit descriptions), (ii) support for a variety of fault models (e.g., faults in logic, and in registers), and (ii) automation of data collection and analysis (e.g., classification into outcome categories and computation of fault propagation paths). The framework is developed as a set of TCL and Perl scripts that extend *Modelsim*. To direct automated operation, the user describes a fault injection experimental campaign (in a text file) by specifying (i) the number of faults to inject, (ii) the duration of an injected fault (for transient faults), (iii) the fault model (e.g., bit flip in the state of a flip-flop or bit flips in the output signals), (iv) a gate-level, HDL description of the studied system, (v) an HDL test bench supplying workload to the system, and (vi) additional parameters such as the simulation time and the clock frequency.

Why gate-level injection. The soundness of a fault injection study conducted in a simulated environment depends on the fidelity with which the simulation captures, both in error-free and erroneous conditions. A fault injection simulation conducted at RTL (e.g., [5]) can only capture the effects of faults in the flip-flop and memory elements; it does not provide enough detail about the actual implementation and the timing of combinational logic. For example, our preliminary experiments indicate that two gate-level implementations (based on carry-ripple adder and carry-look-ahead architecture) of an 8 bit adder from the same RTL design, show remarkably different fault masking in the combinational logic (67% and 41%, respectively). Therefore, to capture the effects of faults in the combinational logic, one must resort to a gate-level simulation, since the resulting error behavior depends on the timing and the topology of the gate-level circuit representation. Based on this observation, we propose the following fault injection methodology:

- Implement a target system (i.e., synthesize, place, and route) on an FPGA device.
- Extract, from the obtained FPGA implementation, a *gate-level* and *post place-and-route* descriptions of the target system, annotated with timing information. In this way, the system timing behavior can be accurately modeled since a logic gate delay is computed based on its capacitive load (depending on the fan-out and length of the interconnections).
- Verify the correctness of the gate-level description by simulation and comparison with the behavior of an RTL description, if the latter is available.
- Use the gate-level description to inject faults while the target system is simulated to execute a representative application.

Although a gate-level simulation can guarantee a sound fault injection study, this approach incurs about an order of magnitude slowdown compared to RTL simulation (according to our measurements). Solutions have been proposed to speed-up a circuit simulation by using FPGA-based hardware emulation [19]. However, hardware emulation captures a system only from a functional perspective and, consequently, cannot be used to study the effects of faults in the combinational logic. Even in the case of errors in flip-flop and memory units, hardware emulation-based fault injection may result in some inaccuracies because it requires the addition of extra hardware instrumentation that emulates the occurrence of faults, which can change the system timing behavior.

5 Results

Analysis of the data from fault injection experiments leads to several important conclusions:

- The average fault manifestation rate is 4% for the studied workloads (for transient faults in combinational logic with duration equal to the clock cycle and bit-flips in flip-flops). Many of the faults that propagate to the processor interface may not be easy to contain as they originate from the processor's control and speculation logic (which is usually unprotected) and from combinational logic (which cannot be protected with generic technique, such as ECC).
- Fault sensitivity of the combinational logic (4.2% for the fault duration of 60ns) is not negligible, even though this figure is 2.5 times smaller than the fault sensitivity of flip-flops (10.4%). The error contribution from combinational logic may well increase with growing complexity and increased clock frequency of future generation processors [2] [5].
- The logic of control and speculation blocks is more sensitive to faults than that of the execution block (4.4%, 12.8%, and 2.4%, respectively, with a fault duration of 60ns). While this result is somewhat dependent on the application, it indicates that control and speculation logic should be consider as a significant source of potential errors and hence would need aggressive protection. One can expect that this problem would worsen for speculation and control logic of more complex microprocessors.
- Analysis of the error impact (crash, fail silent data violation, incomplete execution) at the application level reveals that majority of crashes (45%) and fail-silence violations (40%) originate from errors in the execution block. While the contribution of speculation and control blocks is smaller (the two blocks collectively cause about 34% crashes and 34% fail-silent violations), the percentage is high enough to justify the need for mechanisms to contain those errors. Even more alarming statistics are observed in the case of application incomplete execution, where 69% of cases are due to errors in speculation (34%) and control (35%) blocks. These data show the need for processor-level detection to contain these errors.

Application	Bubble Sort				Prime Number	
Fault Duration	**7 ns**	**15 ns**	**30 ns**	**60 ns**	**30 ns**	**60 ns**
Faults injected	8324	14808	10196	25836	12027	14816
Fault manifested	68	205	242	1135	250	554
Manifestation rate [%]	0.8 +/- 0.2	1.4 +/- 0.2	2.4 +/- 0.3	4.4 +/- 0.2	2.1 +/- 0.3	3.7 +/- 0.3
Average error latency [cycles]	7.6	7	7.1	8.4	16.6	16.7

Table 4: Manifestation rate at the processor interface as function of the fault duration

Fault Duration	7 ns		15 ns		30 ns		60 ns	
	% of Total Errors	Sensitivity	% of Total Errors	Sensitivity	% of Total Errors	Sensitivity	% of Total Errors	Sensitivity
Combinational Logic	86.80%	0.70%	86.30%	1.20%	90.90%	2.20%	93.10%	4.20%
Flip-flops	13.20%	3.70%	13.70%	6.30%	9.10%	7.30%	6.90%	10.40%

Table 5: Breakdown of error contribution from combinational logic and flip-flops for Bubble Sort

• The combinational logic responsible for manipulating data in the processor is highly fault-sensitive. For example, a fault in *Instruction fetch* (in our design) has 19% and 8% chance respectively, to manifest at the processor interface depending on whether a fault hits combinational logic or a flip-flop. High fault sensitivity is observed for the *Bus interface unit*, which shows 44% (for combinational logic) and 67% (for flip-flops) likelihood of a fault to manifest outside the processor boundaries.

5.1 Fault/Error Manifestation

Table 4 summarizes fault injection results for different fault durations, while executing the Bubble Sort application. The data indicate that error manifestation rate increases from 0.8% to 4.4% by varying the fault duration from 7ns to 60ns. Table 4 also provides the average error latency (in terms of clock cycles) defined as the difference between the time instant of the fault injection and the time when the error is manifested at the interface of the processor. The latency is about 7 cycles for shorter fault durations (7ns to 30ns) and increases to about 8 cycles for a fault duration of 60ns.

Results for 30ns and 60ns fault durations while executing a Prime Number application indicate a manifestation rate slightly lower than Bubble Sort (3.7%) and a higher average fault latency (about 17 clock cycles). The longer fault latency is due to the fact that the Prime Number application uses processor registers to store temporary values and, hence accesses memory less frequently than Bubble Sort application. As a result, an error in the Prime Number program can be latent in the processor for a longer time before affecting the processor interface.

5.2 Overall Error Contribution and Fault Sensitivity

Table 5 provides (i) the breakdown of manifested errors according to type of injected component—combinational logic (LUTs and gates) and flip-flops, and (ii) error sensitivity as a ratio between the manifested errors and injected faults for a given component type (i.e., combinational logic or flip-flops). If the injected faults are distributed between the combinational logic and flip-flops according to their overall area occupation, a vast majority (about 90%) of manifested errors come from the combinational logic. On the other hand, flip-flops are much more sensitive to faults, e.g., 2.2% (for combinational logic) versus 7.3% (for flip-flops) for fault duration of 30ns (half the clock cycle).

Table 5 indicates that the sensitivity of the combinational logic increases with the duration of faults (0.7% and 4.2% for fault duration of 7ns and 60ns, respectively). Since the above numbers are dependent on the ratio between fault duration and the clock frequency, the manifestation rate is likely to increase with increasing processor clock frequencies under the assumption that the fault duration remains constant. This is consistent with conclusions in [2] predicting significant increase in soft errors due to faults in combinational logic.

Table 6 reports a breakdown of the results aggregating the data in terms of the functional blocks (each block includes several functional units, see Table 1) comprising the DLX processor. One can see that the speculation block and the control block are more sensitive to faults than the execution block. These results confirm that a protection strategy should not only focus on the errors stemming from the execution block, but it should also consider the impact of faults in speculation logic and in the instruction sequencing logic. The usual statement, "*Speculation is fault tolerant,*" [20] is not always true, since the logic that is in charge of controlling the speculation, e.g., resolve branch or force the rollback of the state, can misbehave and lead to application failures.

5.3 Error Contribution and Fault Sensitivity Break-up per Functional Unit

Table 7 provides (i) the breakdown of manifested errors per functional unit (defined as the percentage of total number of errors coming from each functional unit), (ii) the sensitivity of the combinational logic within a functional unit (defined as the percentage of faults injected in combinational logic belonging to a given functional unit that result in errors), (iii) the sensitivity of the flip-flops in a functional unit (defined as the percentage of faults injected in flip-flops belonging to a given functional unit that result in errors), and (iv) the fault sensitivity of the functional unit (defined as the percentage of faults injected in a given functional unit that result in errors).

Functional Unit	7 ns		15 ns		30 ns		60 ns	
	Manifestation Rate [%]	Sensitivity [%]	Manifestation Rate [%]	Sensitivity [%]	Manifestation Rate [%]	Sensitivity [%]	Manifestation Rate [%]	Sensitivity [%]
Execution	0.3	0.5	0.4	0.6	0.8	1.3	1.5	2.4
Control	0.2	0.8	0.2	1.3	0.4	2.3	0.8	4.4
Speculation	0.0	1.0	0.2	4.0	0.3	6.5	0.6	12.8
Memory interface	0.3	2.6	0.5	4.2	0.8	6.4	1.4	11.1

Table 6: Error contribution and sensitivity of processor functional blocks for Bubble Sort

Fault Duration	7 ns				15 ns				30 ns				60 ns			
Functional Unit	Contribution to error manifestation	CL Sensitivity	FF Sensitivity	Sensitivity	Contribution to error manifestation	CL Sensitivity	FF Sensitivity	Sensitivity	Contribution to error manifestation	CL Sensitivity	FF Sensitivity	Sensitivity	Contribution to error manifestation	CL Sensitivity	FF Sensitivity	Sensitivity
Alu	6	0	5	1	7	1	7	1	8	2	8	2	8	3	11	4
Mdu	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rf	29	1	0	1	22	2	0	2	25	3	0	3	***27***	6	0	6
Cu	2	4	0	3	2	8	5	7	1	4	6	5	1	6	11	7
Bru	2	1	0	1	3	2	5	3	4	5	8	5	3	8	9	8
If	3	1	2	1	9	4	6	5	10	9	4	8	10	***19***	***8***	***16***
Lsu	15	1	11	2	22	4	16	5	17	5	22	7	***13***	9	17	10
Wb	15	2	6	2	5	1	4	1	5	2	5	2	5	3	11	4
Biu	10	5	0	5	13	12	0	12	13	20	0	20	***15***	***44***	***67***	***44***
Rb	6	1	0	1	10	1	13	1	9	2	20	2	7	3	12	3
Dp	13	1	0	1	7	1	5	1	9	3	0	2	11	6	5	6

Table 7: Contribution from each functional unit to the total error count, sensitivity of combinational logic, flip-flops and overall block (all the numbers are normalized and expressed in percentages

Results in Table 7 indicate, that functional units that contribute maximum towards error manifestation are: *Register File* (about 27%), *Bus interface unit* (about 13%), and *Load-store unit* (about 15%) regardless of the fault duration. Units most sensitive to faults are the *Bus interface unit* and *Instruction fetch*. In both cases a fault usually results in an incorrect address being accessed and/or a wrong instruction being executed, which may hang or crash the application.

From a designer's point of view the overall error contribution of a component indicates which functional unit should be protected to reduce the overall error manifestation rate, while the fault sensitivity pinpoints, which unit to target in order to maximize the effectiveness of the protection in relation to the area occupied by the unit. To illustrate this point, let us calculate the area effectiveness of replication-and-voting for two functional units with significantly different fault sensitivities. The protection effectiveness of replicating the unit *U* can be calculated as follows:

Protection_Effectiveness (U) = manifestation rate (U) / area_occupation(U)

This factor gives the decrease in the manifestation rate at the expense of an increase in area occupation. Units with higher protection effectiveness indexes are a better choice for replication. From simple algebra it is possible to see that this number is proportional to the sensitivity.

Scenario 1: *Bus interface unit* (fault sensitivity 44%, for fault duration of 60ns; see Table 7)

Protection_Effectiveness (Biu) = manifestation rate(Biu) / area_occupation(Biu) = 15 / 1.3 = 11.5

Scenario 2: *Register file* (fault sensitivity 6%, for fault duration of 60ns; see Table 7)

Protection_Effectiveness (Rf) = manifestation rate(Rf) / area_occupation(Rf) = 27 / 21.4 = 1.3

Note that, the units with higher sensitivities are also the ones with higher protection effectiveness indexes. Even if the manifestation rate of the *Rf* is higher than that of *Biu*, the latter is definitely a better selection for replication under the area constraint. This relatively simple example illustrates that the fault sensitivity can be a useful measure for deciding which unit to protect (replicate in our example) for maximum benefit under limited resources.

A detailed analysis on fault sensitivity also reveals that the flip-flops are more sensitive than combinational logic, since the flip-flops hold the architectural microstate of the processor. However, there is clear evidence that, for functional units responsible for data manipulation, combinational logic becomes more fault sensitive than flip-flops. For example, the sensitivity of combinational logic versus flip-flops for *Instruction fetch* (19% versus 8% for 60ns fault duration, respectively) and *Dispatcher* (6% versus 5%, respectively) is higher. This can be explained by the fact that these units are in charge of decoding the instructions and generating the signals controlling the data-path through combinational networks. Therefore, a fault in the combinational logic can easily propagate and corrupt the computation.

Fault Duration	30 ns		60 ns	
Functional Unit	CL	FF	CL	FF
Alu	90	10	92	8
Bru	78	22	81	19
Biu	100	0	99	1
Cu	50	50	50	50
Dp	100	0	99	1
If	87	13	88	12
Lsu	73	28	83	17
Mdu	0	0	100	0
Rf	100	0	100	0
Rb	96	5	98	2
Wb	82	18	79	21

Table 8: Error contribution from combinational logic and flip-flops (all the numbers expressed in percentages)

From the data in Table 8, it is evident that the total number of manifested errors contributed by the combinational logic is dominant (50% to 100% of observed errors), under the assumption that the faults in the combinational logic and in the flip-flops are equally probable. While in the current generation processors flip-flops are more likely to upset the system than the combinational circuits (the masking effect), results from Table 8 indicate that the increasing complexity of combinational logic in the future generation processors is likely to change this situation. As a result, for next-generation technologies, using parity or ECC to protect the state of the processor may not be sufficient and more aggressive protection strategies should be deployed, e.g., application aware runtime monitoring of processor's internal behavior (e.g., [24]).

5.3 Error Impact at the Application Level

Table 9 and Table 10 summarize the distribution of manifested errors among the outcome categories. One can see that about 53% of errors that propagate outside the processor boundaries do not impact the correct behavior of the application. The reason is the natural error masking within the application. E.g., an error may cause an incorrect value to be written into a memory location, however, if the subsequent operation overwrites the corrupted location before the corrupted data are used, the error is inconsequential for the correct behavior of the application.

Outcome	% of Errors
Crash	23%
Fail-silent data violation	13%
Incomplete execution	12%
No effect	53%

Table 9: Error impact at the application level

Table 10 shows that, for all the functional units, apart from the *Bus interface unit*, the likelihood of an error to be masked by the application varies between 0.5% and 6.1% for *Reorder buffer unit* and *Load-store unit*, respectively. Much higher figure is observed for the *Bus interface unit* (34%) is due to the application specifics. This is because *Biu* controls the usage of the data bus and consequently errors in *Biu* may often result in loading or storing data from/to incorrect memory location. As long as the application does not attempt to access an illegal location (which would result in a crash), the error may impact outcome of a single iteration causing an intermediate data sort to be incorrect. However, due to the nature of the bubble sort algorithm, the data will be correctly sorted in the following iteration.

Table 10 shows that three units, *Rf, If,* and *Dp* account for 70% of the crashes. As expected, faults in the *If* and *Biu* are most likely to cause crashes (5% sensitivity). This is due to the fact that a fault in *If* can corrupt the address field of an instruction and a fault in the *Biu* can force an access to illegal memory location. Both cases are likely to result in a segmentation fault (or application crash).

About 63% of the fail silent data violations are from three units, *Rf*, *Lsu*, and *Rb*. *Rf,* and *Rb* hold the application state, and consequently, a fault can lead to fail silent data corruption. Similarly, a fault in the *Lsu* can corrupt a value read/written from/to memory. While the application may not crash, it is likely to produce an incorrect result.

Incomplete execution is mainly due to faults in two functional units, *Rb* and *If,* (which collectively contribute to almost 50% of the cases). The data indicate, however, that a fault in the *Cu* is most likely to lead to incomplete execution (13% sensitivity; three times greater than *If* sensitivity 4%). The high sensitivity of the *Cu* can be explained by the fact that a fault in the commit unit could lead to a stall in processor pipeline, although the program-counter continues to be updated. As a result, application never completes.

Table 11 shows the contribution of functional blocks to the different outcome categories. The largest percentage of crashes (45%) and fail-silence violations (40%) originate from errors in the *execution* block. While the contribution of speculation and control blocks is smaller (the two blocks collectively cause about 34% crashes and 34% fail-silent violations), the percentage is high enough to justify the need for mechanisms to contain those errors. It is significant for incomplete executions, 69% of the cases are due to errors in speculation (34%) and control (35%) blocks. Since incomplete execution can be caused, for example, by a stalled *If*, a stalled *Cu* or a corrupted *Rb*, detecting such faults in the processor boundaries may require dedicated, processor-level techniques.

Functional Unit	Crash		Fail Silent Data Violation		Incomplete Execution		No Error (Fault Masking)	
	Contribution from unit	Sensitivity of the block	Contribution from unit	Sensitivity of the block	Contribution from unit	Sensitivity of the block	Contribution from unit	Likelihood of fault masking
Alu	4%	0.4%	11%	0.5%	9%	0.4%	8%	1.7%
Bru	1%	0.8%	5%	1.6%	0%	0.0%	3%	3.2%
Biu	9%	5.1%	5%	1.7%	3%	0.9%	25%	34.2%
Cu	0%	0.0%	0%	0.0%	11%	13.3%	0%	0.0%
Dp	13%	1.4%	8%	0.5%	9%	0.5%	11%	2.8%
If	16%	5.4%	5%	1.0%	23%	4.0%	7%	5.4%
Lsu	9%	1.4%	18%	1.6%	11%	0.9%	16%	6.1%
Mdu	0%	0.0%	0%	0.0%	0%	0.0%	0%	0.0%
Rf	41%	2.2%	29%	0.8%	9%	0.2%	21%	2.5%
Rb	4%	0.4%	16%	0.8%	26%	1.2%	3%	0.5%
Wb	3%	0.5%	3%	0.2%	0%	0.0%	8%	2.9%

Table 10: Outcomes of the fault injection as function of the component injected

Functional Block	Crash	Fail Silent DataViolation	Incomplete Execution
Execution	45%	40%	17%
Control	17%	24%	35%
Speculation	17%	10%	34%
Memory Interface	21%	26%	14%

Table 11: Outcomes of the fault injection as function of the component injected

5.4 Propagation Paths

Analysis of fault/error propagation between the functional units of the processor allows us to (i) obtain a better understanding of how the processor fails and (ii) determine where a detection mechanism should be deployed in order to maximize the chances of detecting a fault before it corrupts the state of the processor or propagates outside the processor.

Generation of error propagation graph. As a first step, a static analysis on the VHDL gate-level netlist of the target processor is conducted to build a *static connectivity diagram*, representing the communication paths between the functional units of the processor. A sample of the static connectivity diagram is depicted in Figure 3a. Nodes labeled *Alu, Rb, Dp, Biu* correspond to the functional units of the processor and the arcs depict unidirectional wires between the units. The procedure used to generate the propagation paths for each injected fault consists of several steps enlisted below.

1. Generate an error propagation table for an injected fault in the form of a time-ordered sequence of functional units affected by the fault (until the error surfaces at the processor interface). An example of a possible time ordered sequence for a fault injected in the *Alu* and spreading into the *Rb, Dp,* and *Biu* is shown in Figure 3b.

2. Compare the constructed error propagation sequences with the static connectivity diagram to eliminate infeasible propagation paths. For instance, comparing the connectivity diagram in Figure 3a, with the time ordered sequence depicted in Figure 3b, two paths $Alu \rightarrow Rb \rightarrow Biu$ (black solid lines Figure 3c) and $Alu \rightarrow Dp$ (a gray dashed line in Figure 3c) are extracted as potential error propagation paths. Faults cannot propagate from *Biu* to *Dp* (no physical path exists) or from *Rb* to *Dp* (unidirectional connection points from *Dp* to *Rb*); consequently, these two paths are discarded as infeasible. Further, a fault cannot propagate through the path from *Dp* to *Rb* due to the timing order of the occurrence of the fault at the interface of *Dp* and *Rb.*

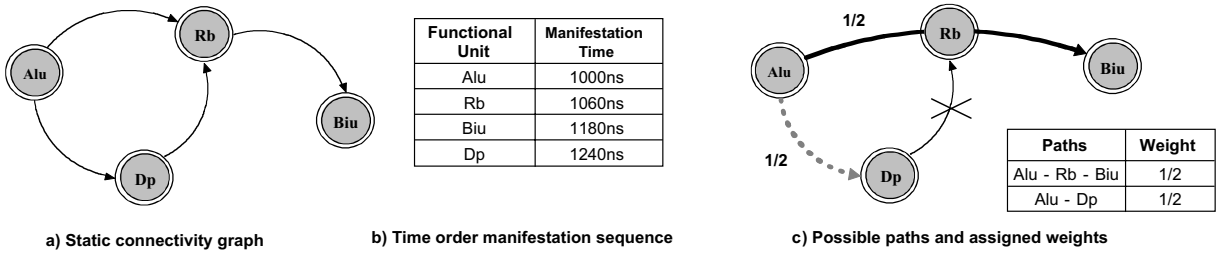

Figure 3: Constructing the list of fault/error propagation paths

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

3. Associate/assign to each arc (of any feasible fault propagation path) a weight, which gives the probability of occurrence of that fault-path. In general, multiple fault paths can originate from the same injected fault as we cannot conclusively determine which path is actually taken. The weight assigned to each arc originating from a given node is inversely proportional to the number of feasible arcs (or fault-paths). In the example scenario in Figure 3b, there are two feasible paths and, hence, the weight assigned to each outgoing arc from the node labeled *Alu* is 1/2. Note that if a particular path has already been traversed by any of the previous faults, the current weight is summed to the previous weight.

4. Collect (feasible) propagation paths for all faults.

5. Create a global list of unique fault/error propagation paths with weights adjusted according to the frequency of occurrence of that particular path.

6. Normalize the final weight of each propagation path with respect to the total number of manifested errors to obtain a global list of all propagation paths and the associated probability of taking the path.

7. Merge the identified paths into a global fault/error propagation graph (global list).

This procedure results in a global list of unique paths with a certain probability of occurrence. Ultimately the most sensitive fault-paths are obtained. Figure 3 depicts a global fault/error propagation paths obtained from the analysis of experimental data from the Bubble Sort workload. Each node in the graph corresponds to one of the functional units. Figures reported along the arcs represent the probability that an error propagates from the source node to the destination node.

Discussion. Analysis of fault propagation paths indicate:

- As expected, the blocks from which faults propagate directly to the *Biu* are the *If* (10%) (which sends the program counter value), the *Lsu* (17%) (which sends the address of the memory location to be fetched by load instructions), the *Wb* (24%) (which sends the address and value of the of the store instruction) and the *Dp* (5%) (which controls the proper operation of the *Biu*).
- The *Dp* and *Rb* are two sources of faults that propagate to the *If*, potentially causing a change in the program counter (PC) value (10% of the faults in *Dp* and 26% of the faults in *Rb* propagate to *If*). The *Dp* forces the change in PC value in case of a misprediction or when there is an exception while decoding an instruction. An exception in the *Alu* (17%), *Mdu* (32%), *Lsu* (10%), or *Bru* (36%) is flagged in the *Rb*, which then propagates to the *If*.
- One of the main reasons for fault manifestation is the corruption of the processor state, i.e. state of the *Rf*. The *Rf* can be corrupted by the *Dp* (20%), the *If* (19%), or the *Cu* (16%). Of the three, *Dp* just resets some of the components of *Rf* in case of an exception, and *If* sends the PC value at various stages of execution. Consequently, one can assume that *Cu* is the main source of corruption of *Rf*.

The most sensitive paths from our analysis are reported in Table 12. Just 20 most-frequent paths account for about 50% of all paths traversed.

Results from fault/error propagation path analysis can be used to devise effective protection strategies. For example, the *Rf* is a complex block since it occupies about 22% of the total area of the processor. Hence, using replication mechanism to detect an error is an expensive technique. From the global error propagation path, it is clear that a fault that is generated in the *Rf* corrupts the *Dp* 30% of the time. The *Dp* gets the contents of the registers from the *Rf* and delivers this value to the *Alu*. Following this path from *Rf* to *Alu*, we can understand how an error can corrupt the computation. In the above mentioned case a detection mechanism can be deployed between the interfaces of the *Rf* and the *Cu* for checking anomalous values of the registers. A possible mechanism would be to use a comparator to check the value of the registers against boundaries computed by profiling the application.

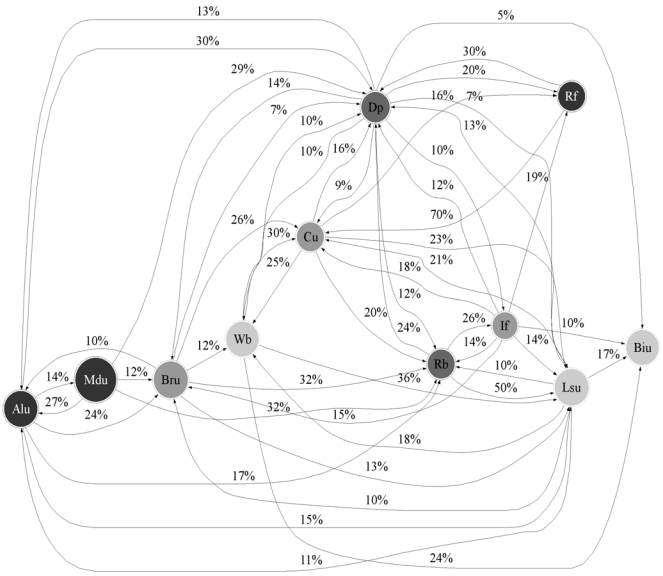

Figure 3: Global fault/error propagation paths

Dp→ Biu	7.30%
Dp → Wb → Biu	5.30%
Dp → Lsu → Biu	4.10%
Dp → Rb → Lsu → Biu	3.60%
Dp → If → Biu	3.60%
Dp → Cu → Lsu → Biu	3.40%
Lsu → Biu	2.50%
If → Biu	2.50%
If → Lsu → Biu	1.90%
Rf → Cu → Wb → Biu	1.80%

Table 12: Most frequent error path propagations

6 Related Work

Techniques to implement HDL-level fault injectors for hardware systems can be classified into two major groups [21]: (i) fault injectors based on modification of the description of the target system to inject faults, such as saboteurs or mutants [22]; (ii) fault injectors relying on changes of the state of the simulation (e.g., the value of variables or signals) through simulator commands, without modifying the system description [23]. While the tools from the first group enable injecting broader range of faults, the injectors in the second group are easier to implement and allow speed-up of the simulation.

Several studies have analyzed a processor behavior using fault injection. In [5] the effects of transient faults in a superscalar Alpha-like processor are studied. Faults are injected into memory elements (flip-flops and RAM) in the RTL description of this processor. The fault model used is a single bit-flip of a state element (equivalent to our fault model for flip-flop for one clock cycle). It is observed that about 88% of injected faults do not manifest, the most error sensitive units are register file and the speculation logic.

In [10] a performance model of a processor is used to estimate the sensitivity of different blocks within the microprocessor to soft errors. The Architectural Vulnerability Factor (equivalent to our soft error sensitivity) is introduced to quantify the error sensitivity. While the methodology is fast and allows simulating (at instruction level) workload with million of instructions, the used performance model limits the sensitivity analysis to estimating only the contribution of the faults from flip-flops in the blocks affecting the performance of the microprocessor.

A fault injection study of PicoJava II is reported in [6]. Single bit-flips are injected into flip-flops at the RTL description of the processor. The error sensitivity analysis indicates that the microcode unit (or control logic) is as sensitive as the execution unit. In [7], a technique to estimate the failure rate of digital designs, implemented on an SRAM-based FPGA is proposed. It is based on the computation of signal probabilities and then of the error propagation probabilities based on the topological structure of the circuit. This technique can compute the system failure probability very fast, but it cannot classify the effect of a fault at the application level.

7 Conclusions

This paper studies the impact of soft errors in a microprocessor for embedded applications. The target system is a gate-level implementation of a DLX-like processor. The synthesized design is simulated, and transient faults are injected to stress the processor while it is executing selected applications. Collected data are analyzed to assess (i) fault sensitivity of control and speculation logic as compared to other functional blocks, (ii) error contribution from combinational circuits versus flip-flops, and (iii) error propagation between the functional units. The results indicate that the fault sensitivity of control and speculation blocks is comparable or even larger than that of execution block. Also, the combinational logic, though less sensitive than flip-flops, could potentially lead to increased error manifestation in future technologies.

8 Acknowledgments

This work was supported in part by NSF grant ACI 0121658 ITR/AP, MURI grant N00014-01-1-0576 and Gigascale Systems Research Center (GSRC/MARCO). We thank Fran Baker for her careful reading of an earlier draft of this paper.

References

[1] T. Karnik, P. Hazucha, J. Patel, "Characterization of soft errors caused by single event upsets in CMOS processes," *IEEE Transactions on Dependable and Secure Computing*, 1(2), 2004.

[2] P. Shivakumar, et al., "Modeling the effect of technology trends on the soft error rate of combinatorial logic," *Proc. of Int'l Conference on Dependable Systems and Networks*, 2002.

[3] T. Juhnke, H. Klar, "Calculation of the soft error rate of submicron CMOS logic circuits," *IEEE Journal of Solid-State Circuits*, 30, 1995.

[4] P. Hazucha, et al., "Measurements and analysis of SER-tolerant latch in a 90-nm dual-V/sub T/ CMOS process," *IEEE Journal of Solid-State Circuits*, 39(9), 2004.

[5] N.J. Wang, et al., "Characterizing the effects of transient faults on a high-performance pipeline," *Proc. of Int'l Conference on Dependable Systems and Networks,* 2004.

[6] S. Kim, A. K. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," *Proc. of Int'l Conference on Dependable Systems and Networks*, 2002.

[7] G. Asadi, M.B. Tahoori, "An analytical approach for soft error rate estimation of SRAM-based FPGAs," *Proc. of MAPLD Int'l Conference*, 2004.

[8] Z. Alkhalifa et al., "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, 10(6), 1999.

[9] R.Karri, B. Iyer, "Introspection: A register transfer level technique for concurrent error detection and diagnosis in data dominated designs," *ACM Trans. Design Autom. Electr. Syst*. 6(4), 2001.

[10] S.S. Mukherjee, et al.: "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *Proc. of MICRO-36*, 2003.

[11] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2002.

[12] H. Eveking, SuperScalar DLX documentation, http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/DlxPdf.zip.

[13] L. B. Freeman, "Critical charge calculations for a bipolar SRAM array," *IBM Journal of Research and Development*, 40(1), 1996.

[14] P. Hazucha, "Background radiation and soft errors in CMOS circuits," *Linkping Studies in Science and Technology, Dissertations*, 2000.

[15] http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/.

[16] G.S. Choi, R.K. Iyer, "FOCUS: Sn experimental environment for fault sensitivity analysis," *IEEE Trans. on Computers*, 41(12), 1992.

[17] Z. Kalbarczyk, et al., "Hierarchical simulation approach to accurate fault modeling for system dependability evaluation," *IEEE Transactions on Software Engineering*, 25(5), 1999.

[18] N. J. Buchanan et al. "Total ionizing dose effects in a Xilinx FPGA," ATLAS-LARG internal note ATL-LARG-99-003, 1999.

[19] K.-T. Cheng, S.-Y. Huang, W.-J. Dai, "Fault emulation: A new methodology for fault frading," *IEEE Transactions on CAD,* 18(10), 1999.

[20] C. Weaver, T. Austin, "A fault tolerant approach to microprocessor design," *Proc. of Int'l Conference on Dependable Systems and Networks*, 2001.

[21] J. Gracia, et al., "Comparison and application of different VHDL-based fault injection techniques," *Proc. of Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, 2001.

[22] J. Boue, et al., "MEFISTO-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance," *Proc.of Int'l Symposium on Fault-Tolerant Computing*, 1998.

[23] V. Sieh, O. Tschache, F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," *Proc. of Int'l Symposium on Fault-Tolerant Computing*, 1997.

[24] N. Nakka, Z. Kalbarczyk, R.k. Iyer, J. Xu, "An architectural framework for providing reliability and security support," *Proc. of Int'l Conference on Dependable Systems and Networks*, 2004.