2011 29th IEEE VLSI Test Symposium

Exponent Monitoring for Low-Cost Concurrent Error Detection in FPU Control Logic

Michail Maniatakos
EE Department
Yale University
michail.maniatakos@yale.edu

Yiorgos Makris
EE & CS Departments
Yale University
yiorgos.makris@yale.edu

Prabhakar Kudva
IBM T. J. Watson
Research Center
kudva@us.ibm.com

Bruce Fleischer
IBM T. J. Watson
Research Center
fleischr@us.ibm.com

***Abstract*—We present a non-intrusive concurrent error detection (CED) method for protecting the control logic of a contemporary floating point unit (FPU). The proposed method is based on the observation that control logic errors lead to extensive datapath corruption and affect, with high probability, the exponent part of the IEEE 754 floating point representation. Thus, exponent monitoring can be utilized to detect errors in the control logic of the FPU. Predicting the exponent involves relatively simple operations, therefore our method incurs significantly lower overhead than the classical approach of duplicating the control logic of the FPU. Indeed, experimental results on the openSPARC T1 processor show that, as compared to control logic duplication, which incurs an area overhead of 17.9% of the FPU size, our method incurs an area overhead of only 5.8% yet still achieves detection of over 95% of transient errors in the FPU control logic. Moreover, the proposed method offers the ancillary benefit of also detecting 98.1% of datapath errors that affect the exponent, which cannot be detected via duplication of control logic. Finally, when combined with a classical residue code-based method for the fraction, our method leads to a complete CED solution for the entire FPU which provides a coverage of 94.4% of all errors at an area cost of 16.32% of the FPU size.**

I. Introduction

As aggressive scaling continues to push technology into smaller feature sizes, various design robustness concerns continue to arise. Among them, the frequent occurrence of transient errors [1] has resurfaced as a contemporary problem of interest. This problem is mainly attributed to strikes by neutrons or alpha particles and the corresponding single event upsets (SEUs) in memory bits, or single event transients (SETs) in combinational logic, which may potentially result in a soft error. However, several other factors such as design marginalities, Negative Bias Temperature Instability (NBTI), coupling, power supply noise, etc. [2], [3] also threaten the robustness of modern microprocessor units. The increasing severity of the above threats has spawned renewed efforts in developing cost-effective concurrent error detection (CED) methods for various key components of a circuit.

Floating point units (FPUs), in particular, are among the most crucial and hardest to protect [4], [5]. And while progress is being made on solutions using error detecting/correcting codes for the datapath portion of an FPU [6], [7], [8], little is known about its control logic, where either duplication [9] or Triple Modular Redundancy (TMR) [10] techniques are usually applied. Control logic errors might have catastrophic impact on the FPU output [11], [12], jeopardizing the application execution and providing the end-user with erroneous results. Furthermore, the size of control logic in modern FPUs is significant, often amounting up to 20% of the FPU size, thus rendering necessary the application of error detection methods.

In this study, we propose an alternative method to protect the control logic of an FPU by monitoring the exponent part of the floating point representation. Our method is based on the conjecture that a control logic error will incorrectly guide the datapath and, by extension, severely alter the expected outcome of the performed operation. As a result, it is highly likely that a control logic error will modify the value of the exponent portion of the floating point output. Given that it is relatively straightforward to calculate the correct exponent through simple operations, monitoring exponent correctness leads to an inexpensive yet very efficient CED method for the FPU control logic. Furthermore, it provides the ancillary benefit of detecting errors in the exponent part of the representation and, when combined with a residue code-based error detection method for the fraction, it results in a very low-cost CED solution for the entire FPU.

The rest of the paper is organized as follows: Section II briefly describes existing techniques for the protection of FPUs. Section III describes the proposed exponent monitoring-based CED method, followed by section IV where the development of the simulation-based experimental infrastructure and the actual CED implementation is presented. The merit figures of the proposed method, namely the attained coverage and incurred overhead, are assessed in section V, followed by conclusions in section VI.

II. Error Detection in FPUs

Several error detection methods have been proposed for protecting FPUs. Most of them, however, target the datapath and have been ported from the corresponding techniques for integer arithmetic, while methods specifically designed to protect the FPU control logic have yet to be developed.

A. Datapath

The most popular technique for reliable arithmetic operations is residue codes [13], [14], [15]. Low-cost residue codes are single arithmetic error detecting codes with unidirectional error detecting capabilities. The efficiency of residue codes depends on the selection of the check base b. The higher the base the more errors the code will detect, yet also the more expensive the hardware overhead which will be incurred. Popular base selections are b = 15 (4 bits) and b = 3 (2 bits). In both cases the resulting *modulo* circuit is highly simplified and the theoretical error detection percentage is $1-(1/2^4)=93.4\%$ for b = 15 and $1-(1/2^2)=75\%$ for b = 3. Residue codes have been successfully applied to various designs [7], [16], [17].

Other techniques include Berger codes [18], [19] and two-rail checkers [20], [16]. Berger codes are optimal unidirectional error detecting codes. ALUs using Berger encoded operands have been shown to be strongly fault-secure [7], [11].

B. Control Logic

The simplest and most straightforward CED solution for random logic, such as the control logic of the FPU, is duplication [9]. The main advantage of duplication is simplicity and applicability to any given design. However, the $> 100\%$ area overhead (including the comparators) and the extra delay required for checking make duplication less appealing for modern FPUs. Furthermore, control in modern, pipelined FPUs is distributed across multiple components, necessitating manual and tedious effort to identify and replicate it.

Triple Modular Redundancy (TMR) [10] has similar properties to duplication, with the added advantage of error correction. However, the hardware overhead of $> 200\%$ (including the voter) makes it prohibitive for commercial designs.

III. Proposed CED Method

Our method is based on the conjecture that an error in the control logic will lead to extensive datapath corruption, which will propagate to the exponent part of the floating point representation. Numerous examples can be provided to show the impact of control errors on the exponent and justify our approach:

- *Special case control:* The control logic identifies whether the input operands are NaN (Not a Number), Infinity, 0, etc. Mishandling of special cases due to errors will result in a completely different output with an incorrect exponent. For example, any operation with NaN results in a NaN. If the control logic mistreats a normal operand as NaN, then the operation 3*5 will result in NaN (exponent = 255) instead of 15 (exponent = 130).
- *Algorithm stage control:* All floating point operations go through several stages before generating the final results. In case a stage is skipped or repeated (e.g. one more or one less division round is performed) due to a control error, the result will be incorrect and is likely to be reflected in the exponent.
- *Select lines:* Control logic is responsible for driving the correct operands to the datapath. In case an error occurs and the control drives a 0 instead of a 7, the operation 7*18 will result in a 0 (exponent = 0) instead of 126 (exponent = 133).
- *Operation control:* Along with the operands, the control logic also drives the signals for the operation selection. Therefore, if due to an error the operation changes, say from addition to subtraction, then the operation 2.0+1.9 will result in 0.1 (exponent of 123) instead of 3.9 (exponent of 128).

These are only a few examples of datapath corruption due to control logic errors, supporting our conjecture that errors in the FPU control logic can be detected by monitoring the datapath. Since the exponent part of the datapath is likely to be affected and fairly simple to calculate [21], we seek to develop a low-cost CED method for the control logic by predicting and verifying the exponent part of the floating point result.

A. Calculating the exponent

In this section, we discuss the exponent calculation for each of the three types of FPU functions, namely arithmetic operations, conversions and other operations. We note that the exponent is calculated independently of the fraction operation, hence the result is not exact since possible fraction normalization may affect the final value of the exponent.

1) Arithmetic operations: The first category is arithmetic operations, such as additions, subtractions, multiplications and divisions. We remind that the IEEE 754 representation of normalized floating point operands is $(-1)^s * 1.f * 2^e$, where s is the sign, f is the fraction and e the exponent. Thus, multiplication and division exponent calculation is simple, i.e.,

$$\begin{aligned}((-1)^{s_1} * 1.f_1 * 2^{e_1}) * ((-1)^{s_2} * 1.f_2 * 2^{e_2}) \\ = (-1)^{s_1+s_2} * 1.f_1 * 1.f_2 * 2^{e_1+e_2}\end{aligned} \tag{1}$$

for multiplication and

$$\begin{aligned}((-1)^{s_1} * 1.f_1 * 2^{e_1})/((-1)^{s_2} * 1.f_2 * 2^{e_2}) \\ = (-1)^{s_1+s_2} * 1.f_1/1.f_2 * 2^{e_1-e_2}\end{aligned} \tag{2}$$

for division. So we simply need to add (for multiplication) or subtract (for division) the exponents, operations which can be performed by the same hardware structure. In case the fraction overflows and needs to be normalized, the exponent needs to be adjusted accordingly. Hence, for arithmetic operations, we can only predict the exponent of normalized results with a ± 1 accuracy. Consequently, if an erroneous result differs from the correct result by 1, error masking will occur. However, our conjecture is that, in the presence of a control logic error, the datapath is corrupted extensively, hence the probability of such masking is very low. Indeed, the results presented in section V corroborate this conjecture.

For addition and subtraction, the exponent is the largest of the two operand exponents, therefore a simple comparator suffices to calculate it (similar to the multiplication/division cases, normalization may be required). This does not apply in the case of cancelation (i.e., when there is a subtraction of operands with equal exponents or exponents that differ by 1). In this case, the exponent can take a wide range of values and cannot be computed accurately without information from the fraction. In order to moderate cost, our CED method taps into the existing FPU hardware in order to obtain this information (rather than replicating it), hence error masking due to common mode failures may occur. Nevertheless, as we show in the results Section V, such masking is very small.

2) Conversions: Another common operation performed in FPUs is conversion from/to integer/floating point representations. The exponent of the results can be exactly calculated by appropriately offsetting the input operand. For floating point precision conversions (single to double and vice versa), the exponent needs to be offset by ± 896, since the actual exponent is $e_s - 127$ in single precision and $e_d - 1023$ in

TABLE I
OPENSPARC T1 FLOATING POINT INSTRUCTIONS

Operation	Result Exponent	Fraction Normalization
Addition	$max(e_1, e_2)$	Yes
Subtraction	$max(e_1, e_2)$[1]	Yes
Multiplication	$e_1 + e_2$	Yes
Division	$e_1 - e_2$	Yes
Single to Double	$e_1 + 896$	No
Double to Single	$e_1 - 896$	No
Integer to Single	$MSB(i1)$ + 127	No
Integer to Double	$MSB(i1)$ + 1023	No
Negation	e_1	No
Absolute Value	e_1	No

[1] Equal or different-by-1 exponents may lead to cancelation.

double precision. Thus, for single to double conversion, the exponent is $e_d = (e_s - 127) + 1023$ and for double to single $e_s = (e_d - 1023) + 127$. For integer conversions, the exponent is a function of the most significant bit. Table I summarizes the exponent operation for different FPU functions.

3) Other operations: Modern FPUs usually implement more operations, such as absolute value, negation and comparison. In all these operations, the exponent is very simple to calculate. Negation/absolute value operations affect only the sign (i.e., the exponent is the same). Comparison operation results are implementation specific, as the output result is the comparison result and not a floating-point number. For example, SPARC ISA defines the exponent field of the output as 0, and the comparison result is stored in the flags field.

IV. EXPERIMENTAL SETUP

In order to assess the effectiveness of our method we built an extensive simulation infrastructure to perform error injection experiments. Since our target is transient errors, we need to perform a large number of injections; therefore, the infrastructure must support very fast simulations and error impact evaluation.

A. Test Vehicle

The test vehicle of our study is the register transfer-level (RTL) model of the openSPARC T1 microprocessor [22], the open source version of the UltraSPARC T1 microprocessor. The openSPARC T1 processor has eight SPARC processor cores which have full hardware support for four threads. Each SPARC CPU core can send a packet to the shared Floating Point Unit (FPU), using the cache-processor crossbar (CPX). Conversely, the FPU can send a packet to any one of the eight cores using the processor-cache crossbar (PCX). A floating point instruction is delivered from the cores to the FPU in either one- (single operand instructions) or two-packet transfer. One source operand is transferred in each cycle and the crossbar always provides a two-cycle transfer. In case of single operand instructions, an invalid transfer is produced in the second cycle [23].

Since the FPU is a single shared resource, each of the eight SPARC cores may have a maximum of one FPU instruction waiting to be executed. Thus, the FPU can hold up to 8 instructions at a given time. The FPU implements the SPARC V9 floating-point instruction set, and is fully compliant with the IEEE 754 standard [24]. The floating point register file and floating point state register are in the SPARC core Floating point Front-end Unit (FFU), which is unique for every core (and not shared like the FPU).

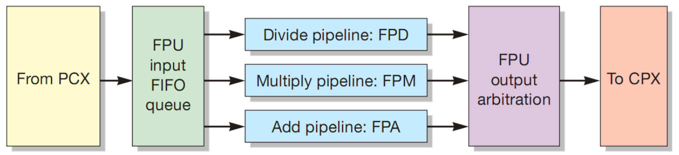

Fig. 1. T1 FPU Functional Block Diagram [23]

The FPU includes three execution pipelines:

- Floating-point adder (FPA): Executes additions, subtractions, comparisons and conversions
- Floating-point multiplier (FPM): Executes multiplications
- Floating-point divider (FPD): Executes divisions

Incoming instructions are stored in a 16 entry x 155 bit FIFO queue (unless the FIFO is empty, in which case it is bypassed). In each cycle, one instruction may be issued from the FIFO and one instruction may complete and exit the FPU. Fig. 1 shows a block diagram with the three independent pipelines. Not-a-Number (NaN) source propagation is supported by steering the NaN source through the pipeline to the result.

B. CED implementation

The FPU is a shared resource with multiple floating-point instructions in flight. Moreover, the latency of some floating-point instructions (i.e. division) is variable and cannot be predicted a priori. Hence, it is not possible to predict which instruction should exit the FPU at each time. Instead, the exponents calculated for each incoming instructions are stored in an array which is indexed using the CPU ID of the outgoing instruction. The CPU ID is a unique identifier because each of the 8 cores can have a maximum of one outstanding FPU instruction. A thread with an outstanding FPU instruction switches out while waiting for the FPU result. This allows up to 8 instructions to be in the FPU. Therefore, storing the signatures requires a memory with 8 entries.

The block diagram of the CED implementation for the openSPARC T1 is presented in Fig. 2. This implementation applies to any pipelined FPU that executes multiple floating point instructions, such as the IBM PowerPC 405 FPU, Intel Pentium FPU and the SPARC T2. In case an FPU executes only one instruction at a time, the memory array is not needed and the output result can be checked directly.

The ± 1 exponent component presented in the diagram is needed for the arithmetic operations that may require fraction normalization, as explained in Section III-A1.

C. Experiment flow

Fig. 3 shows the data flow of our experiments. First, we use a python-based assembly generator which we developed to generate multi-threaded (MT) assembly utilizing floating point instructions. This synthetic workload is then simulated in the openSPARC T1 environment using `sims`, and Value Change Dump (VCD) traces are collected at the input of the FPU. These traces are then converted to a separate testbench

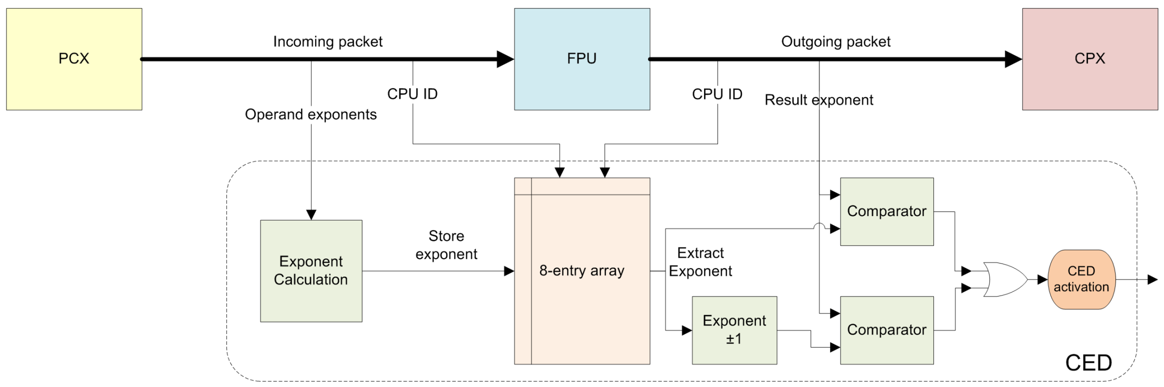

Fig. 2. Block Diagram of Proposed CED Structure

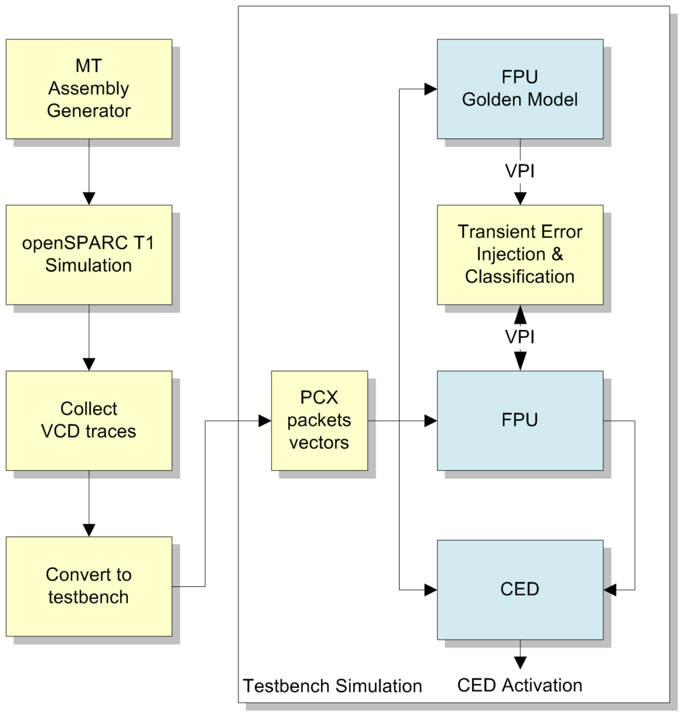

Fig. 3. Simulation Infrastructure

using Synopsys `vcat`. This testbench can be simulated using Synopsys `vcs` without the need to simulate the entire microprocessor model, leading to a 10x simulation speed-up.

The assembly generator can generate up to 32 different assembly files, one for each thread. The user can specify the percentage of the floating-point instructions in the file, as well as the desired number of instructions for each pipeline (FPA, FPM, FPD). Floating point registers are randomly selected for each instruction. 10 of the registers contain special values (NaNs, Inf, 0) to ensure operations with special numbers. Transient error injection is performed during simulation by mutating the microprocessor model for one clock cycle using the parallel saboteurs technique. An extensive description of the RTL error injection method can be found in [25].

The transient error injection is controlled by a python script through Verilog Procedural Interface (VPI) calls. The same script is used for error classification, with the help of a golden model that runs in parallel with the injected FPU model.

D. Hardware synthesis

In order to provide hardware overhead estimates, we synthesized the FPU model using Synopsys Design Compiler targeting a 90nm library. The timing constraints were set to a clock period of 1GHz, to match the running speed of the UltraSPARC T1. The total area of the synthesized FPU is 816,660μm^2 (587,947μm^2 combinational, 181,110μm^2 non-combinational and 47,601μm^2 net interconnect area). Figure 4 shows the hierarchy of the FPU along with the area percentage of each main module. The `*_CTL` and `*_DP` blocks represent the pure control and the datapath portion of each module, respectively. The largest module is the 54x54 multiplier. The division pipeline is rather small (and, naturally, rather slow at the same time). The model also contains a few more very small modules, such as repeaters (to optimize timing) and scan-control modules, which are not shown in the figure. These modules along with the pipeline registers and the wiring add up to the remaining 23.3% of the FPU area. Overall, the control logic amounts to 16.1% of the FPU size.

V. Experimental Results

This section describes the experimental results that support our conjectures. We simulated the openSPARC T1 microprocessor using two different types of synthetic workload: The first one (`FP-100`) consists of 100% floating point operations, to resemble applications with intense need for floating point calculations. The second (`FP-1`) consists of 1% floating point instructions, matching the profile of common applications that place very little demand on the FPU. On average, `FP-100` had 5 floating point instructions in the FPU (either executing or queued) and a maximum of 8 (one floating point instruction from each core). In contrast, `FP-1` had an average of 1 and a maximum of 4 floating point instructions in the FPU. For each of the two workloads, 5 million transient errors were injected, uniformly distributed over time and location across the FPU.

A. FPU Error Impact Analysis

The first set of results, shown in Table II, present statistics regarding the impact of injected errors on the FPU output. As expected in a transient error injection campaign, masking is very high; indeed, among the injected errors, only 2.13% for `FP-100` and 1.45% for `FP-1` reach the FPU output (i.e. non-masked errors). `FP-1` has fewer non-masked errors since

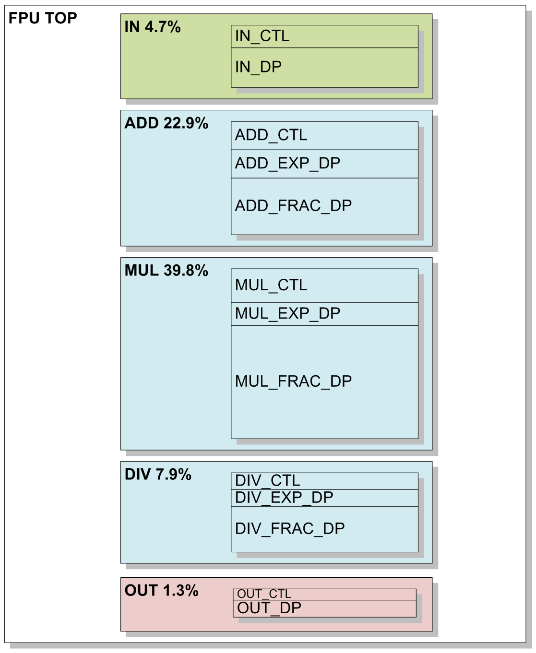

Fig. 4. FPU Hierarchy and Area Breakdown

fewer instructions flow through the FPU at the same time, thus increasing the chance for a transient error to strike on an inactive part. However, the distribution of non-masked errors is similar for the two different workloads.

The key observation from these results is that the average non-masked error rate of control modules (_CTL) for FP-100 is 9.2% (i.e., 1 out of 10 transients affect the FPU), a percentage that is is much higher than the average non-masked rate of datapath modules for the same workload, which is only 1.63%. In other words, errors in the control logic are six times more likely to affect the output than errors in the datapath. This supports our claim that, despite the control logic being smaller (i.e., 16.1% of the FPU size), protecting it is necessary for ensuring reliable FPU operation.

We also point out that the percentage of non-masked errors varies among the different control modules, with `in_ctl`, `div_ctl` and `out_ctl` having higher percentages. This is expected for the `in_ctl` and `out_ctl` modules, since all instructions have to go through them and they always contain critical information regarding instruction execution. As for `div_ctl`, division instructions have 10 times more latency than other instructions (up to 61 cycles), so the probability that the FPD pipeline will be occupied by a valid instruction during the workload execution is much higher.

B. Exponent Monitoring vs. Duplication for Control Logic

The second set of results, shown in Table III, compare the proposed exponent monitoring method to traditional duplication for performing CED in the FPU control logic.

In terms of area overhead, the cost of duplication is 17.9% of the FPU size, of which 16.1% is to replicate the control logic and 1.8% is to compare. In contrast, the proposed exponent monitoring incurs only one third of this cost, for a total of 5.8% of the FPU size. Both methods operate in parallel with the FPU and do not cause noticeable delay overhead.

TABLE II
ERROR CLASSIFICATION STATISTICS

	FP-100		FP-1	
Module	**# of Injected Errors**	**Non-Masked Errors**	**# of Injected Errors**	**Non-Masked Errors**
fpu_top	495,585	2.78%	509,648	2.01%
in	86,501	3.97%	89,888	2.79%
in_ctl	33,117	12.13%	34,066	10.97%
in_dp	163,318	2.78%	167,595	1.87%
add	54,646	1.43%	55,874	1.28%
add_ctl	92,062	2.04%	94,832	1.76%
add_exp_dp	105,283	0.44%	108,368	0.57%
add_frac_dp	567,209	0.59%	582,034	0.43%
mul	75,006	1.98%	77,442	1.31%
mul_ctl	59,829	3.71%	61,907	2.81%
mul_exp_dp	54,652	2.29%	55,953	1.48%
mul_frac_dp	2,885,318	1.60%	2,961,476	0.89%
div	36,552	7.70%	37,228	5.51%
div_ctl	52,266	11.93%	53,915	9.53%
div_exp_dp	27,038	4.50%	27,421	3.34%
div_frac_dp	182,919	4.45%	189,131	3.83%
out	42,125	4.74%	43,234	3.49%
out_ctl	7,052	16.42%	7,229	15.83%
out_dp	86,758	4.37%	89,132	3.06%
Total	**5,107,236**	**2.13%**	**5,246,373**	**1.45%**

TABLE III
EXPONENT MONITORING VS. DUPLICATION

FPU Control CED Method	Area Overhead	Coverage of		
		Control	Exponent	Fraction
Duplication	17.9%	100%	-	-
Monitor Exponent	5.8%	95.1%	98.1%	15%

In terms of effectiveness, exponent monitoring detects 95.1% of control logic errors, as opposed to the 100% coverage provided by duplication. The remaining 4.9% is attributed to control logic errors which only affect the fraction portion of the result and never propagate to the exponent, as we explained in section III-A1. However, exponent monitoring provides the ancillary benefit of also covering 98.1% of the errors that affect the exponent, which control logic duplication is unable to detect. Once again, the small error masking of 1.9% is because of the ± 1 comparison and because of cancelation, as explained in section III-A1. The implication of this ancillary benefit is that no additional CED method is needed to cover the exponent portion. These results corroborate our conjecture that most errors in the control logic will result in an incorrect exponent and support the cost-effectiveness of our exponent monitoring-based CED method for FPU control logic.

C. Cost-Effective CED for Entire FPU

The last set of results examines the utility of exponent monitoring in developing a cost-effective CED method for the entire FPU. We note that, as shown in Table III, exponent monitoring also detects around 15% of the errors that only affect the fraction portion of the floating point representation, which the duplication method is unable to detect. Yet this percentage is very small, hence additional steps need to be taken in order to provide a complete FPU CED solution. To this end, we investigate how our method can be combined with a base-15 residue code for the fraction, in order to reduce the overall CED cost for the FPU. Specifically, we compare three alternative scenarios: (i) using base-15 residue codes for the

TABLE IV
COMPARISON OF CED SOLUTIONS FOR ENTIRE FPU

Detection Method			Coverage				Hardware
Control	Exponent	Fraction	Control	Exponent	Fraction	Total	Overhead
-	Res-15	Res-15	0%	94.3%	94.3%	**59.2%**	**14.82%**
Duplication	Res-15	Res-15	100%	94.3%	94.3%	**96.2%**	**29.78%**
Exponent Monitoring		Res-15	95.1%	98.1%	94.3%	**94.4%**	**16.32%**

fraction and the exponent but leaving the control unprotected, (ii) adding control logic duplication to the above solution, and (iii) combining exponent monitoring with base-15 residue code only for the fraction (since the exponent is already covered).

The results reported in Table IV demonstrate two key points: First, if control is left unprotected, the overall fault coverage would be a mere 59.2%. This shows, once again, that protection of control logic is necessary in modern FPUs. Second, the proposed exponent monitoring method enables a complete FPU CED solution that provides almost equivalent coverage to the duplication-based solution (i.e., 94.4% vs. 96.2%), yet incurs almost half of the cost (i.e., 16.32% vs. 29.78%), thereby constituting a very appealing option.

VI. CONCLUSION

We presented a novel method for detecting transient errors in the control logic of a modern FPU. We demonstrated that errors in the control logic lead to an extensive corruption of the datapath and, by extension, have a high probability of affecting the exponent field of the operation output. Therefore, independently calculating and validating the exponent of the outgoing packet provides very high coverage to such errors. As demonstrated on the openSPARC T1 processor, the proposed exponent monitoring-based CED method costs less than one third of the cost of duplicating the control logic, while maintaining over 95% of its coverage. Moreover, in conjunction with a known residue code-based method for the fraction of the floating point representation, it facilitates a complete CED solution which offers over 94% coverage for the entire FPU at the cost of 16.32% of its size, which to our knowledge, constitutes the most cost-effective approach to date.

REFERENCES

[1] Y. Savaria, N.C. Rumin, V.K. Agarwal, and J.F. Hayes, "Soft-error filtering-A solution to the reliability problem of future VLSI digital circuits," in *IEEE Proceedings*, 1986, vol. 74, pp. 669–683.

[2] C. Metra, M. Favalli, and B. Ricco, "On-line detection of logic errors due to crosstalk, delay, and transient faults," in *International Test Conference*, 1998, pp. 524–533.

[3] T. Karnik, P. Hazucha, and J. Patel, "Characterization of soft errors caused by single event upsets in cmos processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[4] J. Gaisler, "Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications," in *IEEE International Symposium on Fault-Tolerant Computing*, 1994, pp. 128–130.

[5] A. Naini, A. Dhablania, W. James, and D. Das Sarma, "1 GHz HAL SPARC64R Dual Floating Point Unit with RAS features," in *IEEE Symposium on Computer Arithmetic*, 2001, pp. 173–183.

[6] P. Eibl, A. Cook, and D. Sorin, "Reduced precision checking for a floating point adder," in *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, 2009, pp. 145–152.

[7] J.C. Lo, "Reliable floating-point arithmetic algorithms for error-coded operands," *IEEE Transactions on Computers*, pp. 400–412, 1994.

[8] S.M.H. Shekarian, A. Ejlali, and S.G. Miremadi, "A Low Power Error Detection Technique for Floating-Point Units in Embedded Applications," in *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2008. EUC'08*, 2008, vol. 1.

[9] TEMIC, "TSC692E Floating-point Unit User's Manual for Embedded Real Time 32 bit Computer (ERC32)," 1996.

[10] W.L. Gallagher and E.E. Swartzlander, "Fault-tolerant Newton-Raphson and Goldschmidt dividers using time shared TMR," *IEEE Transactions on Computers*, pp. 588–595, 2000.

[11] J.C. Lo, S. Thanawastien, T.R.N. Rao, and M. Nicolaidis, "An SFS Berger check prediction ALU and its application toself-checking processor designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 4, pp. 525–540, 1992.

[12] G.G. Langdon and C.K. Tang, "Concurrent error detection for group look-ahead binary adders," *IBM Journal of Research and Development*, vol. 14, no. 5, pp. 563–573, 1970.

[13] A. Avizienis, "Arithmetic algorithms for error-coded operands," *IEEE Transactions on Computers*, vol. 22, no. 6, pp. 567–572, 1973.

[14] E. Kinoshita, H. Kosako, and Y. Kojima, "Floating-point arithmetic algorithms in the symmetric residue number system," *IEEE Transactions on Computers*, vol. 100, no. 23, pp. 9–20, 1974.

[15] A. Sasaki, "The Basis for Implementation of Ad idive Operations in the Residue Number System," *IEEE Transactions on Computers*, vol. 100, no. 17, pp. 1066–1073, 1968.

[16] D.A. Anderson and G. Metze, "Design of totally self-checking check circuits for m-out-of-n codes," *IEEE Transactions on Computers*, vol. 100, no. 22, pp. 263–269, 1973.

[17] A. Avizienis, G.C. Gilley, F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin, "The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers*, vol. 100, no. 20, pp. 1312–1321, 1971.

[18] J.M. Berger, "A note on error detection codes for asymmetric channels," *Information and Control*, vol. 4, no. 1, pp. 68–73, 1961.

[19] M.A. Marouf and A.D. Friedman, "Design of self-checking checkers for Berger codes," in *IEEE International Symposium on Fault-Tolerant Computing*, 1978, vol. 8, pp. 179–184.

[20] M. Nicolaidis, "Self-exercising checkers for unified built-in self-test (UBIST)," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 3, pp. 203–218, 1989.

[21] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[22] Sun Microsystems, "OpenSPARC T1 specifications," http://www.opensparc.net/opensparc-t1/index.html.

[23] Sun Microsystems, "OpenSPARC T1 Microarchitecture Specification," 2006.

[24] D. Stevenson et al., "IEEE standard for binary floating point arithmetic," *ACM SIGPLAN Notices*, vol. 22, no. 2, pp. 9–25, 1987.

[25] M. Maniatakos, N. Karimi, A. Jas, Tirumurti, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Transactions on Computers (TCOMP)*, 2010 (to appear).