Vol.2012-ARC-199 No.5
2012/3/27

Achieving Effective Fault Tolerance in FU array by Adding AVF Awareness

Tanvir Ahmed ,[†1] Jun Yao [†1]
and Yasuhiko Nakashima[†1]

Fault-tolerance now plays an important role to cover the increasing soft/hard error rates in electronic devices along the advances of process technologies. Error detection with negligible performance impedance and low hardware overhead is accordingly a main concern to keep efficient high dependability. In this paper, a fault-tolerable FU array is proposed with the awareness of architectural vulnerable factor (AVFs). Specifically, we designed a method to help fast locate the erroneous execution in FU array by effectively checking the most vulnerable branches of the data path. It can be further applied to detect multi-site faults or to reduce less important redundancy.

1. Introduction

With the improvement in the CMOS technology, size of the semiconductor devices is shrinking rapidly, which leads to many advantages in modern microprocessor design like low power consumption, low manufacturing cost, high operating frequency, and high density of transistors. Conversely, these advantages affect the noise margins, susceptibility to transient faults, and make the electronic devices more vulnerable to defects. Specifically, high energy radiation particles and electromagnetic interference produce a high rate of soft errors. Similarly, transistor and interconnect reliability wore, due to the shrinking transistor technology and over heating. Therefore, it is essential to embed fault-tolerance in microprocessor to get a correct calculation as well as detect the permanent faults at run-time especially in an environment which may contain many fault attacks.

Different techniques have been proposed for protecting microprocessors against faults. Error correcting codes are best suited for memory structure than the processor core, as their area, performance, and power overhead. On the other hand, redundant execution on the processor core shows better performance. A number of redundant techniques have been proposed to detect and recover from soft error [1,2]. Error detection by duplicated instructions for super-scalar processor has been proposed in [1], where all the instructions are duplicated and checked thereafter. Similarly, fault-tolerant functional unit (FU) array architecture, EReLA, has been proposed in [2]. Different levels of redundancy have been proposed for this architecture to provide soft and hard error protection. In order to detect the soft errors, all the instructions are duplicated but only the final result is checked before commit to the memory. Therefore, this scheme suffers from a long latency to detect the error occurred at the early stage of the data flow graph. Besides, If there is no additional check instruction inside the data path, eventually the probability of error will accumulate to a large value. When the probability of error finally turns into an actual error, it is not possible to determine which part of the data flow graph is erroneous and a through test will be required to locate the erroneous point, as shown in paper [3]. Many FUs are thus required for detecting permanent failure, which increase the hardware size and potential permanent error vulnerabilities

In this paper, we propose an approach for detecting and recovering the error for FU array based on redundant execution by adding AVF awareness. As this scheme use multiple check instructions on the data flow graph, the detection will be faster and overhead for permanent fault detection is smaller than the approaches proposed in [2,3]. EReLA [4] framework has been used for this study.

The rest of this paper is organized as follows. In Section 2, we gives the definition and calculating method of vulnerability factor of each instruction from the view of permanent error. We then discuss our proposed algorithm and permanent fault detection in Section 3. Section 4 shows the results of the proposed approach, and Section 5 concludes the whole paper.

2. Calculation of vulnerability factor

Paper [5] has stated that various programs will respond differently to the same fault rate, according to their different architectural vulnerability factors (AVFs). AVF gives a measure of the probability that a fault will lead to a visible error.

†1 Nara Institute of Science & Technology

Given that the soft event upset occurs in a certain memory block, it will become an error only when latter calculation depends on this faulty block. Accordingly, the AVF depends on the program behaviors, as described in [5].

Similarly, we are using the idea of vulnerability factor in this research to selectively add data verification instructions. We extend the above AVF from soft error only consideration to our purposed permanent fault field. In this research, we treat that the permanent fault vulnerability factor is linear to the gate number inside the functional unit. For example, a 1-bit AND operation requires two 2-input NAND gates, while a 1-bit XOR operation uses four 2-input NAND gates. As a result, the lifespan of the XOR will be relatively shorter than the AND unit under a given gate defect ratio. Applying the consideration to arithmetic operations, the vulnerability factor to permanent defect may be even larger due to the large size and complex wire interconnections inside the arithmetic operations. For example, a 1-bit full adder takes fifteen 2-input NAND gates to finish the calculation. Similarly, a large word-length multiplication uses several stages of adder chains and partial product generator. It is both weak to single error transient (SET) faults and permanent defects because of its large hardware area and relatively long data path. A fault is likely be propagated to the latch after it to become a visible error.

Accordingly, we give the vulnerability factors (VFs) to permanent defects in **Table 1**. In detail, the area results and their corresponding vulnerable factors of logical, arithmetic, and media instructions of our baseline ISA used in this research are given in Table 1. Specifically, we treat that AND operation has a vulnerability factor of 1%. The other vulnerability factors are thus calculated by $1\% \times \frac{Area_{op}}{Area_{AND}}$. As discussed before, logic operations are relatively less complex in hardware and their VFs are thus relatively small. The arithmetic instructions take medium VFs except the very large multiplication unit whose VF reaches 25%. The media operations are combination of logic and arithmetic ones and thus tend to show large VFs. Finally, for LOAD instruction, it has been assumed that the memory is protected with error correcting code (ECC) so that the loaded data can be regarded as error free. The only vulnerability in LOAD comes from the address calculation part which is same to the ADD operation.

The STORE operation is originally designed to take checked data before real commitment, by additionally put a check instruction before it. Its VF is thereby 0%.

Table 1 Vulnerability factor of operations

Operation Type	Operations	No. of Gates	Vulnerability Factor (%)
LOGIC	AND	176	1
	OR	176	1
	XOR	208	1
	SLL/SRL	140	0.75
ARITHMETIC	ADD	892	5
	SUB	1022	5
	MUL	5130	25
	SLA/SRA	372	2
MEDIA	SRL	792	5
	BYTE-HALF	219	1
	SUML/H	996	5
	HALF BYTE	854	5
	SAD	2970	19
	UADD	1320	10
	USUB	1398	10
	MUL	2569	15
MEMORY	LOAD	892	5

3. Proposed algorithm

In this research, the above permanent fault vulnerability factor (VF) has been taken into account to indicate the suitable position to place additional data check instructions for a fast detection of defects. This section introduces the proposed algorithm in Section 3.1 and the corresponding permanent defects location method in Section 3.2.

3.1 Adding check according to VF

Table 1 shows the vulnerability factor of each instruction. Assuming that each operation takes two source operands and gives one result, we can calculate the probability of the error of the result as follows:

$$\Pr(\text{out}) = 1 - (1 - \Pr(\text{s1}))(1 - \Pr(\text{s2}))(1 - \Pr(\text{op})) \quad (1)$$

Pr(s1) and Pr(s2) are the probability of error in the source operands, while Pr(op) is the error probability comes from the operation itself. It can be imaged that the Pr(op) has a direct connection to the vulnerability factor in Table 1. Assume

that the whole data path starts from some checked input value which has 0% probability of error. The output of the first operation will have a probability of error regarding to the operation itself. The latter dependent data will inherit this probability of error, and adds new probability when the data goes forwards through the data flow graph. Although the values of vulnerability factor in Table 1 are actually much larger than a practical probability of error, we are still directly using these values as Pr(op) in the latter parts of this paper to introduce the idea. By this means, we are able to tag the results with the probability of permanent error inside the whole data path.

In our research, we are trying to divide the long data path into different segments. Thus, when some permanent error has occurred in a certain division of the data path, it is possible to search a relatively small space to locate the permanently defected unit. The above tagging of permanent error probability actually gives a good way to balancing the division. In our algorithm, we use a predetermined threshold of probability error for this purpose. Firstly, the probability of error of a data is calculated from Eq.1, by taking inherited error possibilities from source operands and operation itself. When the probability exceeds the threshold, an additional check instruction will be introduced along the data path. The check instruction removes the probability from the checked data by making a determination of its correctness. The data path can thus be divided into several segments with similar error probabilities.

Fig. 1 gives a detailed illustration of this algorithm. For simplicity, we assume that the threshold of error probability is 8%. The data flow graph starts by taking inputs R1, R2, R3 and R4 from register file or memory, which are previously checked results and protected by ECC. It ends by committing final result R6 into the memory. Basically, every operation will be doubly executed and the final result R6 will be compared before committing.

In Fig. 1, the VF of each operation is shown beside the operation. Accordingly, by using our proposed method, we can get the error probabilities along the data flow graph, as show beside each destination register in Fig. 1. After the second stage, both R1 and R2 get error probabilities that exceed the threshold. The check instruction is thus added to make a fast determine whether or not an error happens there. This also makes zone1 and zone2, as shown in Fig. 1.

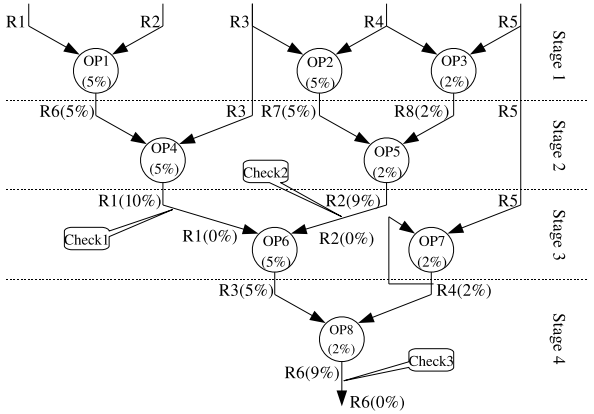

Fig. 1 Data flow graph for proposed algorithm.

It is possible that the data path will take backward data bypassing like operation `R4+=R5`, as shown before the OP7 in Fig. 1. Considering the data path represents a loop kernel, operation `R4+=R5` takes its first operands from the register file in the first iteration and update itself afterward. It is easy to understand that the first iteration, the correctness possibility of result of OP7 is the multiplication value of the correctness probabilities of first R4, R6 and OP7 itself. From the second iteration, using Bayes' theorem, we can have $\Pr(R4) = \Pr(R5\text{correct}, \text{OP7correct} \mid \text{input}R4\text{correct}) \times \Pr(\text{input}R4\text{correct})$. Because the input R4 now comes from the output of the 1st iteration, its correctness already means the R5 and OP7 are safe from the permanent defects. Thus $\Pr(R5\text{correct}, \text{OP7correct} \mid \text{input}R4\text{correct}) = 1$. Accordingly, the permanent probability of R4 is fixed and calculated as in Fig. 1.

3.2 Locating Permanently Defected Units

Normally, dual modular redundancy (DMR) mode is used to check whether or not all the operations are correctly finished along the data path. When the check instructions detect very frequent errors, permanent failure may have occurred in the occupied units. Additional mode will be used to locate the permanently defected unit, as has been introduced in paper [3]. Paper [3] additionally add maximum numbers of check instructions to help understand the erroneous parts

DMR			Locating Permanent Error		
op1	OP1		op1	OP1	
op2	OP2		op2	OP2	chk-op1
op3	OP3		op3	OP3	chk-op2
op4	OP4		op4	OP4	chk-op3
op5	OP5		op5	OP5	chk-op4
op6	OP6		op6	OP6	chk-op5
op7	OP7		op7	OP7	chk-op6
op8	OP8		op8	OP8	chk-op7
		chk-op8(X)			chk-op8(X)

(a) Proposed in paper [3].

DMR			Locating Permanent Error		
op1	OP1		op1	OP1	
op2	OP2		op2	OP2	
op3	OP3		op3	OP3	chk-op2
op4	OP4		op4	OP4	chk-op3
op5	OP5	chk-op4	op5	OP5	chk-op4
op6	OP6	chk-op5(X)	op6	OP6	chk-op5(X)
op7	OP7		op7	OP7	
op8	OP8		op8	OP8	
		chk-op8(X)			chk-op8(X)

(b) Proposed approach.

Fig. 2 Locating Permanently Defected Units.

in the data path, as shown in **Fig. 2(a)**. With the help of additional check instructions according to VF, it is possible to remove some check instructions as shown in **Fig. 2(b)**, given the situation that in DMR mode, CHK-op5 is the first to report error. The detailed location algorithm is introduced in paper [3].

4. Results

We tried our algorithm on several loops as F1, F2, F3, unsharp from an image filter program. **Fig. 3** gives the calculated check instruction numbers when set respectively the error probability threshold to 8%, 10%, 12%, 15% and 20%. Specifically, threshold 20% will add 12 check instructions and divide programs into segments of 4 operations in average. In this way we can reduce the cost under permanent failure location.

Another advantage of the proposed method is that it can help discard part of the duplicated values of the DMR data path after the check instructions have been used. It can help to save some additional power. The evaluation of this part will be studied in detail in future work.

5. Conclusion

This paper proposed an approach to achieve fast permanent failure detection in an FU array by adding AVF awareness. Specifically, check instructions are added selectively according to the error probability along the data path. As a result, permanent fault detection requires less number of functional units than the previous approach. When setting threshold to 20%, by adding 30% checking instructions to segment programs, only 4% of operations are need to be explored after a possible permanent failure.

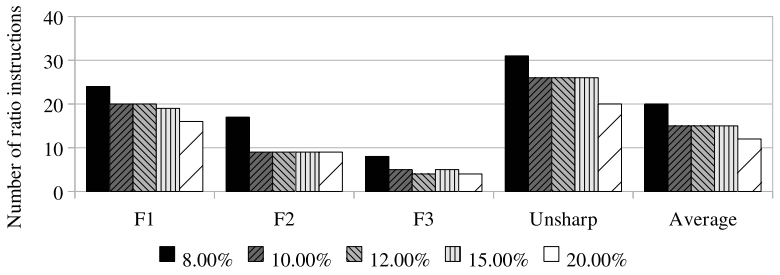

Fig. 3 Number of check instructions w.r.t vulnerability factor threshold for different loops.

Acknowledgments This work is supported by VLSI Design and Education Center (VDEC), University of Tokyo with the collaboration with Synopsys Corporation. This work is supported by JST ALCA, JST A-STEP (FS) No. AS232Z02313A, and Grant-in-Aid for Young Scientists (B) No. 23700060.

References

1) Oh, N., Shirvani, P. and McCluskey, E.: Error detection by duplicated instructions in super-scalar processors, *IEEE Transactions on Reliability*, Vol.51, No.1, pp.63–75 (2002).
2) Oue, S., Yoshimura, K., Yao, J., Nakada, T. and Nakashima, Y.: High Redundancy Instruction Mapping Scheme on FU Array Accelerator, *IPSJ SIG Notes*, Vol.2011, No.19, pp.1–7 (2011).
3) Hazama, Y., Yao, J., Nakada, T., Nakada, T. and Nakashima, Y.: A DMR based Parmanent Error Locating Method for a Dependable FU Array, *IEICE Tech. Rep.*, Vol.111, No.328, pp.47–52 (2011).
4) Yao, J. and Nakashima, Y.: Exploiting Efficiency of Redundant Executions on an FU Array, *IPSJ SIG Notes*, Vol.2011, No.9, pp.1–5 (2011-03-03).
5) Mukherjee, S., Weaver, C., Emer, J., Reinhardt, S. and Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp.29 – 40 (2003).