

A Low-Cost Correction Algorithm for Transient Data Errors

Aiguo Li, Bingrong Hong

School of Computer Science and Technology

Harbin Institute of Technology, Harbin 150001, China

liaiguo@hit.edu.cn

Introduction

Scaling of very large scale integration (VLSI) technologies, coupled with increased integrated circuit complexity, will strongly increase the occurrence of transient faults (also known as soft errors) [1]. Transient faults, unlike manufacturing or design faults, do not occur consistently. Instead, these faults are caused by external events, such as electromagnetic interferences, power glitches, or highly energized particles striking the chip. These events do not cause permanent physical damage to the chip, but can alter signal transfers or stored values and thus cause incorrect program execution [2].

Generally, hardware or software redundancy can be

introduced to counter these transient faults. But the former is expensive, since it requires replicated hardware modules or developing custom hardware equipped with error detection mechanisms that verify operation correctness on line. When development cost is a major concern, as in low volume applications, designers tend to adopt commercially available hardware, even in the case of safety-critical applications. In this context, software-based fault tolerance is an attractive solution, since it allows implementing dependable systems without incurring the high costs.

Transient faults induced by hardware have an impact on software running on it, which causes either control flow errors or data errors in software. In this paper, we mainly focus on data errors, which are errors that affect the values of data variables, registers, or memory locations used by the application. To address this kind of faults, the proposed approaches rely on information redundancy to store multiple copies of the same information and on the introduction of consistency checks whose purpose is to verify the coherence of the replicated information. Representative solutions can be found in served references

[1,3-5]. But most of these approaches proposed so far are not intended for tolerating errors. Indeed, when a error is detected, the approaches either abort the computation or call an external error recovery procedure. As a result, they do not provide any support in masking the effects of detected errors. The other class techniques for tolerating such errors are the recovery block (RB) [6] approach, N-version programming (NVP) [7], some Error Correcting Codes (ECCs) (e.g. Parity bits, Hamming Code, BCH) and some improved fault tolerance methods [8-9] that are capable of detecting and correcting few bit errors. These methods have either long error latency or large time consume. In this paper, another method called Single Bit error Correction (SBC) is proposed to address data flow errors, which can not only detect but also correct the transient errors. A comparison of the proposed method with some other existing methods reveals that SBC has more high error correction rate and lower performance overhead.

Proposed approach

The basic concept of SBC is to generate an extra code information (to be called Accompanied Variable) from every

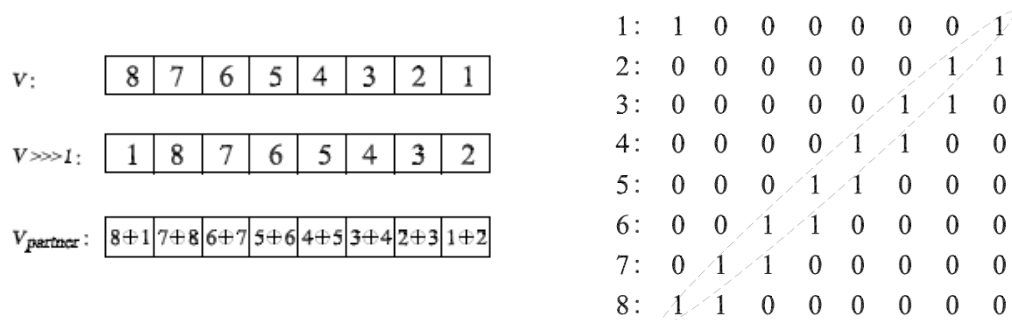
variable in the program. And the accompanied variable must be updated every time a new value is assigned to the variable. Two parameters are needed for the decoding procedure: the variable's value and the accompanied variable's value. In the case of the corruption of one of these two parameters as a consequence of a bit flip, the decoding procedure will detect and correct it.

Suppose there is a variable v , and its accompanied variable encoded with SBC method is $v_{partner}$, then $v_{partner}$ can be gotten through the following formula:

$$v_{partner} = (v \ggg 1) \oplus v \quad (1)$$

where, \ggg is cyclic-right-shift operator, \oplus is exclusive-or operator.

Here, we will take a byte variable as an example to illustrate



the encoding procedure of SBC. Suppose, the bits of the byte variable v are labeled with 1, 2, 3, ..., 8 from the low bits to the high bits, then the results of $v \ggg 1$ and $v_{partner}$ are like this illustrated in Fig. 1.

Fig. 1: The encoding principle of SBC method Fig. 2: The fault matrix of SBC method

Let there is only one bit fault in variable v or $v_{partner}$. If the variable v come into being a bit fault (for example bit 1), which makes v into v' , then the accompanied variable $v'_{partner}$ encoded by SBC from v' will have two bits (for example bit 1 and bit 8) differing from the fault-free variable $v_{partner}$. So, if bit 1 of the byte variable v has a bit-flip, $v'_{partner}$ making XOR with $v_{partner}$ will get the first line of Fig.2, and bit 2 error will get the second line, and so on. All these lines are composed of a matrix, called Fault Matrix, as illustrated in Fig.2. On the other hand, if a bit-flip fault occurs in the accompanied variable $v_{partner}$, the Fault Matrix will have only minor diagonal set to 1, the other elements set to 0, as the dot line indicates in Fig.2.

Algorithm:

According to the fault matrix, we can get SBC's decoding algorithm:

- (1) Input the variable v and its accompanied variable $v_{partner}$.
- (2) Re-compute v 's accompanied variable, which is then XOR'ed with the input $v_{partner}$, and the result is stored in $errCode1$.
- (3) If $v_{partner}$ equal to 0, go to step (8); else continue.
- (4) Correct v using $errCode1$, that is $tmp = v \oplus errCode1$.
- (5) Compute tmp 's accompanied variable, which is then XOR'ed with $v_{partner}$ and the result is stored in $errCode2$.
- (6) Get the value $error = errCode1 \& errCode2$.
- (7) If $error \neq 0$, the variable $v = error \oplus tmp$; else if $error = 0$, $v_{partner} = v_{partner} \oplus errCode1$.
- (8) Return.

Discussion

This algorithm consists of two parts; one is the fault detection part (step 1-step3), the other is fault correction part (step 4-step 7). We will show how this algorithm works through a simple example. Suppose a byte variable v , whose accompanied variable is $v_{partner}$, has a bit-flip fault in

bit 1, then all the bits of the $errCode1$ will equal to the first line of the fault matrix according to step 1 and step 2 in the algorithm. Here, because $errCode1 \neq 0$ (indicating there is a fault happened), the fault has been detected by step 3 and will be corrected in the following steps. In step 4, the variable v making XOR with $errCode1$ will correct its fault in bit 1, but at the same time introduce another fault into bit 8; that is to say, the value of tmp equals to that of the variable v with a fault in bit 8. In the next step, it is similar with step 2 and the $errCode2$ will be the last line of the fault matrix. Then all the bits of the value of variable $error$, gotten in step 6 through making $\&$ operation between $errCode1$ and $errCode2$, will be 0 except bit 8, and whether the value of $error$ equals to 0 or not will be a foundation to judge which value of v and $v_{partner}$ has a fault in step 7. If $error \neq 0$, the variable v is considered of the fault one in this algorithm and is corrected with $error \oplus tmp$, which will clear the fault in bit 8. So far, the fault in bit 1 of variable v has been corrected completely. For the fault in the accompanied variable $v_{partner}$, the correction procedure is similar with above.

Thus, wherever the single bit-flip fault occurs, which may

occur in variable v or its accompanied variable $v_{partner}$, this algorithm is able to detect it and correct it.

To assess the efficiency of the proposed approach, we have performed several fault injection sessions on four simple benchmark programs: Bubble sort of 100 elements, Quick sort of 100 elements, 40x40 matrix multiplication and a fast Fourier transform. At the same time, the comparative analysis between SBC, Hamming code and Duplicating variable with correction (DVC) techniques [10] are also introduced.

The experiment result shows that a general characteristic of all three hardening techniques is the good detection capability for faults affecting the program data area, especially for data section, they were able to detect 99.8% of the injected bit-flips; while for stack section, they also achieved detection more than 90%. As only single bit faults were injected in theory all the faults should be detected and corrected; in our case correction fails can be explained by assuming that those faults corrupting memory bytes where the three method's code intermediate values are stored.

However, the result shows that the SBC method has the highest relative-correction rate (corrected/detected) about 100% for data section and 82% for stack section averagely. Moreover, as far as the absolute detection and correction rate is concerned, the SBC method (13 ~ 53% and 10 ~ 53%) is superior to the other considered techniques. Thus it can be seen that SBC gains an advantage over Hamming code and DVC technologies for detection and correction capabilities of single bit-flips in program data area, especially for the correction ability.

The major drawback of error detection by software means come from the resulting memory area overhead and the increase in execution time for the modified codes. In the matter of the memory area, the overhead factor scope of the three methods is 1.5 ~ 1.9 averagely, and DVC method is slightly bigger than the other two methods. Concerning execution time, the factor of DVC and SBC method is averagely 3.8 and 5 separately, however, Hamming code method entail a very high time overhead, which is about 115 times of SBC's. So, for execute time, SBC and DVC techniques are more better than Hamming code. And SBC

method has a greater error correction capability than DVC method at a cost of a limited increase in performance overhead.

Conclusion

In this paper, a new software-implemented approach for tolerating the effects of transient faults is presented. The major novelty of the approach over the available alternative is the possibility of not only detecting but also correcting transient faults efficiently before they results in program errors. Fault detection and correction are obtained by attaching an extra info to every variable (not duplication only) to which when write operation happens and checking the coherency between them when read operation happens to the variable.

Results issued from fault injection experiments suggests that the method can decline the undetected incorrect output from the original programs's 27% ~ 49% to 0.01% ~ 0.02% for program's data section faults, with almost 100% correction rate; furthermore, for stack section faults, the undetected incorrect output is decreased from 10% ~

18% to 1% ~ 3%, with more than 82% correction rate. Compared with previous techniques, the SBC method has the characteristics of both easy implementation and low memory overhead with a very high fault detection and correction capability.

Acknowledgements

The National Aeronautical Pre-research Program of 'Tenth Five-Year-Plan' under Grant Nos.417010402 has supported this work.

References

- [1] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip faults," *IEEE Transactions on Nuclear Science*, vol. 51(6), 2004, pp. 3510-3518
- [2] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift-software implemented fault tolerance," in *Proc. of the Int. Symposium on Code Generation and Optimization(CGO'05)* , 2005, pp. 243-254.
- [3] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M.

Violante, "Soft-error detection through software fault-tolerance techniques," in DFT'99:IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Austin, USA, November 1999, pp. 210-218.

[4] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results," in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03), Munich, Germany, March 3-7 2003.

[5] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51(1) , 2002, pp. 63-75.

[6] B. Randell, "Design - Fault Tolerance," in *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, and J.-C. Laprie, eds., Springer- Verlag, Vienna, 1987, pp. 251-270.

[7] A. Avizienis, "The N-Version Approach to Fault-Tolerant Systems," *IEEE Trans. Software Engineering*, vol. SE-11, No. 12, Dec. 1985, pp. 1491-1501.

[8] Goutam Kumar Saha, "A Single-Version Scheme of Fault Tolerant Computing," *Intl. Journal of Computer Science*

& Technology, Vol. 6 (1), pp 22-27, 2006, USA.

[9] Goutam Kumar Saha, "Software Implemented Fault Tolerance Through Data Error Recovery," ACM Ubiquity, Vol. 6 (35), ACM Press, 2005, USA.

[10] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new software-based technique for low-cost fault-tolerant application," in *IEEE annual Reliability and Maintainability Symposium*, 2003, pp. 25-28.

Author's Biography

Aiguo Li, received the B.S. degree (2000) and M.S. degree (2003) in Heating, Gas-Supplying, Ventilating & Air Conditioning Engineering from Harbin Institute of Technology, China. He is currently a PhD candidate at Robot Center in the Department of Computer Science of Harbin Institute of Technology. His research interests include design and assessment of dependable software in space computer, fault injection method etc.. He can be reached via liaiguo@hit.edu.cn.

Bingrong Hong, Professor in school of computer science and technology, Harbin Institute of Technology. His research

interests include multi-agent, robots, embedded real time system, fault tolerance and so on.

Ubiquity -- Volume 7, Issue 22

<<http://www.acm.org/ubiquity/>>