To Improve Register File Integrity against Soft Errors By using self - Immunity Technique

A. Praveen
Post Graduate Scholar, Indur Institute of Engineering & Technology, Siddipet

N.Sai Kumar
Assistant Professor, Indur Institute of Engineering & Technology, Siddipet

Abstract - Continuous shrinking in feature size, increasing power density etc, increase the vulnerability of microprocessors against soft errors even in terrestrial applications. The register file is one of the essential architectural components where soft errors can be very mischievous because errors may rapidly spread from there throughout the whole system. Thus, register files are recognized as one of the major concerns when it comes to reliability. The paper introduces Self-Immunity, a technique that improves the integrity of the register file with respect to soft errors. Based on the observation that a certain number of register bits are not always used to represent a value stored in a register.

The paper deals with the difficulty to exploit this obvious observation to enhance the register file integrity against soft errors. We show that our technique can reduce the vulnerability of the register file considerably while exhibiting smaller overhead in terms of area and power consumption compared to state-of-the-art in register file protection. For embedded systems under stringent cost constraints, where area, performance, power and reliability cannot be simply compromised, we propose a soft error mitigation technique for register files.

Key words: self immunity Technique, Register, ECC, MTBF

I.INTRODUCTION

In the early days of computers, “glitches” were an accepted way of life. Since then, as computers have become more reliable (and more relied upon), glitches are no longer acceptable – yet they still occur. One of the most intractable sources of glitches has been the transient “bit-flip”, or soft memory error: a random event that corrupts the value stored in a memory cell without damaging the cell itself. Soft errors* in electronic memory were first traced to alpha particle* emissions from chip packaging materials. Since then, memory manufacturers have eliminated most alpha particle sources from their materials, changed their designs to make them less susceptible. Tests and standards have been developed to measure and improve the resistance of memory chips to alpha particles – but soft errors have not disappeared. Further testing, mostly performed by avionics and space organizations, pinpointed a more pernicious source of soft errors: cosmic rays*. At ground level, cosmic radiation is about 95% neutrons and 5% protons. These particles can cause soft errors directly; they can also interact with atomic nuclei to produce troublesome short-range heavy ions. Cosmic rays cannot be eliminated at their source, and effective shielding would require meters of concrete or rock. To eliminate the soft memory errors that are induced by cosmic rays, memory manufacturers must either produce designs that can resist cosmic ray effects or else invent mechanisms to detect and correct the errors. Over the last decade, and in spite of the increasingly complex architectures, and the rapid growth of new technologies, the technology scaling has raised soft errors to become one of the major sources for processor crashing in many systems in the nano scale era. Soft errors caused by charged particles are dangerous primarily in high atmospheric, where heavy alpha particles are available. However, trends in today’s nanometer technologies such as aggressive shrinking have made low-energy particles, which are more superabundant than high-energy particles, cause appropriate charge to provoke a soft error. Due to their large number of components, supercomputers are particularly susceptible to soft errors.

I.I Register file soft errors

Register file (RF) is extremely vulnerable to soft errors, and traditional redundancy based schemes to protect the

RF are prohibitive not only because RF is often in the timing critical path of the processor, but also since it is one of the hottest blocks on the chip, and therefore adding any extra circuitry to it is not desirable. Device scaling trends dramatically increase the susceptibility of microprocessors to soft errors. Further, mounting demand for embedded microprocessors in a wide array of safety critical applications, ranging from automobiles to pacemakers, compounds the importance of addressing the soft error problem. The paper addresses this challenge by introducing a novel technique, called Self-Immunity to improve the resiliency of register files to soft errors, especially desirable for processors that demand high register file integrity under stringent constraints.

Our contributions within this paper are as follows:

(1) We present a technique for improving the immunity of register files against soft errors by storing the ECC in the unused bits of a register.

(2) We solve the problem of the area and power overhead that typically comes as a negative side effect in register file protection by achieving high area and power saving With a slight degrading in the register file vulnerability reduction (7%) compared to a full protection scheme.

II. SOFT ERROR BACKGROUND AND TERMINOLOGY

II.I.IMTBF and FIT

Vendors express an error budget at a reference altitude in terms of Mean Time between Failures (MTBF). Errors are often further classified as undetected or detected. The former are typically referred to as *silent data corruption* (SDC); we call the latter *detected unrecoverable errors* (DUE).For example, for its Power4 processor-based systems, IBM targets 1000 years system MTBF for SDC errors, 25 years system MTBF for DUE errors that result in a system crash, and 10 years system MTBF for DUE errors that result in an application crash. One FIT specifies one failure in a billion hours. Thus, 1000 years MTBF equals 114 FIT (109 / (24×365×1000)). A zero error rate corresponds to zero FIT and infinite MTBF. Designers usually work with FIT because FIT is additive, unlike MTBF.The effective FIT rate for a structure is the product of its raw circuit FIT rate and the structure's *vulnerability factor*, i.e., an estimate of the probability that a circuit fault will result in an observable error. The overall FIT rate of the chip is calculated by summing the effective FIT rates of all the structures on the chip.Current predictions show that typical raw FIT rate numbers for latches and SRAM cells vary between 0.001 – 0.01 FIT/bit at sea level

II.II Vulnerability Factors

The effective FIT rate per bit is influenced by several *vulnerability factors* (also known as *derating factors or soft error sensitivity factors*). In general, a vulnerability factor indicates the probability that an internal fault in a device's operation will result in an externally visible error. If the latch is accepting data 50% of the time, this effect results in a*timing vulnerability factor* for the latch of 50%.The earliest schemes of register file protection such as Triple Modular Redundancy (TMR) and ECC can achieve a high level of fault tolerance but they may not be suitable solutions in embedded systems due to their power and area overhead The proposed approach in utilizes the Cross-parity check as a method for correcting multiple errors in the register files. Spica et al.showed that there is a very little gain (just 2%) in fault tolerance for caches if they increase the protection to Double Error Correction while the overhead for that gain is considerable. Building on the concept of Architectural Vulnerability Factor (AVF), introduced by Mukherjee, Yan et al. proposed the *Register Vulnerability Factor* (RVF) to describe the likelihood that a soft error in registers can be spread to other system parts. In general, a value is written into a register, then it is read frequently, and later a new value is written again. Thus, any soft error occurring during "write-write" or "read-write" intervals will have no effect on the system, because it will be corrected automatically by the next write operation. On the other hand, "write-read" and "read-read" intervals are considered vulnerable intervals as is depicted in Figure-2.1. The RVF of a register is defined as the sum of the lengths of all its vulnerable intervals divided by the sum of the lengths of all its lifetimes.

$$Vulnerability(reg) = \frac{\sum Vulnerable\ Interval\ Time}{\sum LifeTime}$$

Finally, the total vulnerability of the register file is assumed as the sum of vulnerability of all registers.

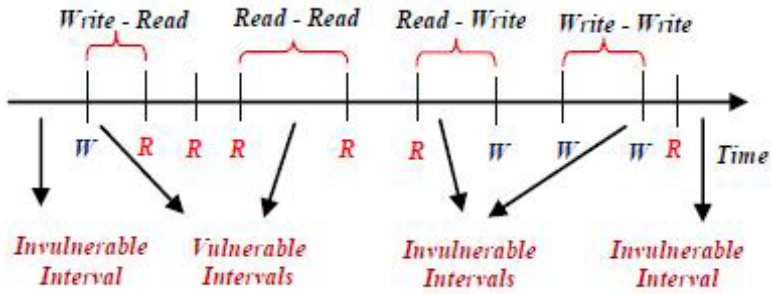

Figure-2.1:Different Register Access Intervals

The pure software approach at compile level introduced by Yan et al.I re-schedules the instructions in order to decrease the RVF of a register file but the proposed technique is not always very effective because it may increase the execution cycles and even the RVF in some benchmark

II.III Register Vulnerability factor

Toward the goal to measure register file susceptibility to soft errors accurately and quantitatively, we define the register vulnerability factor (RVF) to be the probability that a soft error in registers can be propagated to other system components (i.e., functional units, memory). As depicted in figure-2.1,In this below Equation, RVi represents any register value, the Susceptible Time (RVi) represents the time intervals that RVi is exposed to the susceptible intervals (i.e., W-R and R-R intervals for RVi), and the Lifetime (RVi) represents the lifetime of RVi, which is time interval between the time that a register is allocated for RVi and the time it is overlapped by another value. Since both the Susceptible Time (RVi) and Lifetime (RVi) can be easily obtained from a performance simulator, it would be straightforward to compute the RVF.

$$RVF = \frac{\sum SusceptibleTime(RV_i)}{\sum Lifetime(RV_i)}$$

The RVF indicates the probability that register soft errors can spread to other hardware elements and thus impact the system output. The higher the RVF, the lower the register file reliability, and hence more expensive techniques are needed to fight soft errors.In contrast, traditional software optimizations mainly focus on performance. Therefore, the RVF allows compilers to consider both performance and reliability to optimize the register access patterns. Such a software based approach has no hardware overhead, which is fundamentally different from traditional space redundancy or information redundancy techniques.

Techniques to reduce register vulnerability factor are:

(a) Reschedule Instructions to Reduce RVF.

(b) Reliability oriented Register Assignment with Partial ECC Protection..

III.SELF IMMUNITY TECHNIQUE

III.I.Proposed Self-Immunity Technique

We propose to exploit the register values that do not require all of the bits of a register to represent a certain value. Then, the upper unused bits of a register can beexploited to increase the register's immunity by storing the corresponding SEC Hamming Code without the need for extra bits. The Hamming Code is defined by k, the number of bits in the original word and p, the required number of parity bits (approximately log2k). Thus, the code word will be (k + log2k + 1). In our proposed technique, the optimal value of k is the value which guarantees that w, the bit-width of the register file, can cover both k, the required number of bits to represent the value, and the corresponding ECC bits of that value. In other words, the value and its ECC should be stored together within the bit-width of a register. Consequently, the following condition should be valid (k + log2k + 1 ≤ w). Thus, the optimal value of k is 26 in 32-bit architectures and 57 in 64-bit architectures.For instance, when studying 32-bit architectures, where each register can represent a 32-bit value, we may exploit the register values, which require less than or equal to 26 bits by storing the corresponding ECC bits in the upper unused six bits of that register to enhance the register file immunity against soft errors1. We call this technique Self-Immunity and we call such values "26-bit" values. On the other hand, we call register values which need more than 26 bits to be represented "over-26-bit" register values. Figure-3.1 shows the percentage of register values usage for different applications of the MiBench Benchmark compiled for MIPS architecture.

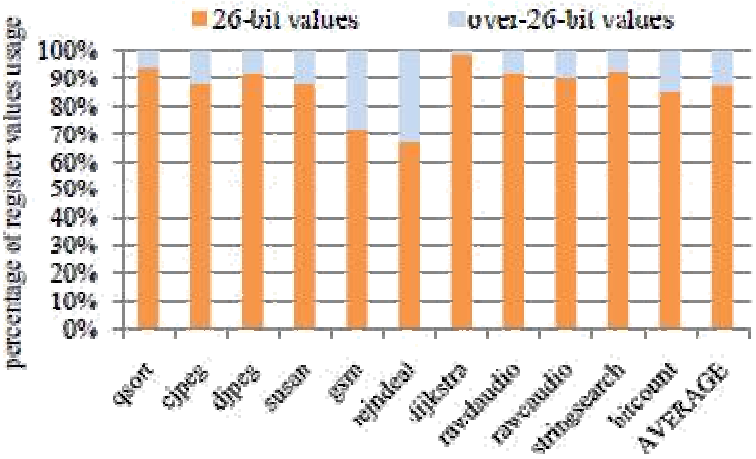

Figure-3.2: "26-bit" register values and "over-26-bit" register values in different benchmarks

As it can be noticed, in all benchmarks most of the register values are "26-bit" values. In other words, the upper six bits of 88% of the stored data in the register file are actually unused. In addition to the previous key observation, the contribution of "26-bit" register values in the total vulnerable intervals is much more than the contribution of "over-26-bit" register values. In Figure3.1, the fraction of vulnerable intervals of each benchmark is reported. As is demonstrated, the fraction of vulnerable intervals of "26-bit" values is 93% on average.

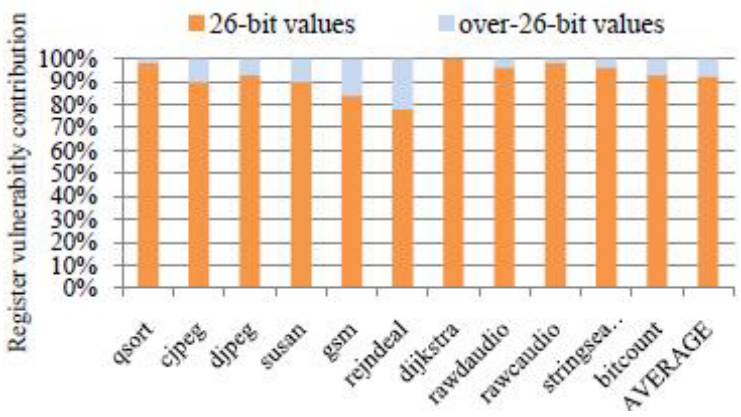

Figure-3.2: The fraction of vulnerable intervals of "26-bit" register values and "over-26-bit" register values in different benchmarks.

III.II Implementation details

Since the probability of multiple bit-errors is largely lower than the single bit-error, a single bit-error model has been considered. In our fault injection environment, faults are injected on the fly while the processor executes an application. In each fault injection simulation, one of the 32 registers is selected randomly and a bit in that register is chosen randomly and then flipped. Notice that a write operation clears out the previous injected error into that register. Likewise, by using a uniform distribution, a random cycle is chosen as the time that soft error occurs. This makes sure that the faults will be injected only when the program is executed. Since an injected fault might produce an infinite loop, a watchdog timer was implemented for the required number of execution cycles. We stop the simulation when the cycle count exceeds two times the number of cycles in the fault-free case.

Towards evaluating our proposed technique, we use different applications from MiBench Benchmark compiled for MIPS architecture to take into account different possible scenarios for register utilization. Simulations were conducted using the MIPS model simulator. When a simulation terminates, the corresponding output information (final results, content of the register file, execution time and state of the processor) are stored and used to classify the simulation.

1. *Wrong Answer*: The application terminates normally but the results produced are not correct.
2. *Latent*: The application terminates normally, the results are correct but at the end of simulation the content of the register file is different from that of fault-free case.
3. *Effect-Less*: The application terminates normally, the results are correct, and the content of the register file is similar to that of fault-free case.
4. *Exception*: The processor detected the injected fault and generated an exception (e.g., invalid address exception).
5. *Timed-Out*: The application failed to terminate and produce results with a predefined time limit.
6. *Stalling*: The processor computed the expected results in a time greater than the time of fault-free case.
7. *Crashing:* The processor fails to terminate normally.

Each benchmark was simulated 10,000 times. As a result, 10,000 soft errors were injected randomly in the register file. This number complies with those used by other research to keep the total time of simulations reasonable. For a fair comparison, we consider three models of the processor:

IV.CONCLUSION

For embedded systems under stringent cost constraints, where area, performance, power and reliability cannot be simply compromised, we propose a soft error mitigation technique for register files. Our experiments on different embedded system applications demonstrate that our proposed *Self-Immunity* technique reduces the register file vulnerability effectively and achieves high system fault coverage. Moreover, our technique is generic as it can be implemented into diverse architectures with minimum impact on the cost. It can be concluded that this technique achieves the best overall result compared to state-of-the-art in register file vulnerability reduction

REFERENCES

[1] Greg Bronevetsky and Bronis R. de Supinski, "Soft Error Vulnerability of Iterative Linear Algebra Methods," in the 22nd annual international conference on Supercomputing, pp. 155-164, 2008.
[2] J.L. Autran, P. Roche, S. Sauze, G. Gasiot, D. Munteanu, P. Loaiza, M. Zampaolo and J. Borel, "Real-time neutron and alpha soft-error rate testing of CMOS 130nm SRAM: Altitude versus underground measurements," in ICICDT'08, pp. 233–236, 2008.
[3] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in International Symposium on Microarchitecture (MICRO-36), pp.29-40, 2003
[4] T.J. Dell, "A whitepaper on the benefits of Chippkill-Correct ECC for PC server main memory," in IBM Microelectonics division Nov 1997.
[5] S. Kim and A.K. Somani, "An adaptive write error detection technique in on-chip caches of multi-level cache systems," in Journal of microprocessors and microsystems, pp. 561-570, March 1999.
[6] G. Memik, M.T. Kandemir and O. Ozturk, "Increasing register file immunity to transient errors," in Design, Automation and Test in Europe, pp. 586-591, 2005.
[7] Jongeun Lee and Aviral Shrivastava, "A Compiler Optimization to Reduce Soft Errors in Register Files," in LCTES 2009.
[8] Jason A. Blome, Shantanu Gupta, Shuguang Feng, and Scott Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in CASES '06, pp. 421–431, 2006.
[9] P. Montesinos, W. Liu, and J.Torrellas, "Using register lifetime predictions to protect register files against soft errors," in Dependable Systems and Networks, pp. 286–296, 2007.
[10] M. Rebaudengo, M. S. Reorda, and M.Violante, "An Accurate Analysis of theEffects of Soft Errors in the Instruction and Data Caches of a Pipelined Microprocessor,"in DATE'03, pp. 602-607, 2003