Exploiting Existing Copies in Register File for Soft Error Correction

Exploiting Existing Copies in Register File for Soft Error Correction

Abdulaziz Eker, Oguz Ergin
Dept. of Computer Engineering, TOBB University of Economics and Technology, Ankara, Turkey
{aeker,oergin}@etu.edu.tr

Abstract—Soft errors are an increasingly important problem in contemporary digital systems. Being the major data holding component in contemporary microprocessors, the register file has been an important part of the processor on which researchers offered many different schemes to protect against soft errors. In this paper we build on the previously proposed schemes and start with the observation that many register values already have a replica inside the storage space. We use this already available redundancy inside the register file in combination with a previously proposed value replication scheme for soft error detection and correction. We show that, by employing schemes that make use of the already available copies of the values inside the register file, it is possible to detect and correct 39.0% of the errors with an additional power consumption of 18.9%.
Index Terms—microprocessor architecture, register file, soft error

1 Introduction

TRANSIENT errors caused by particle strikes pose a significant challenge to designers because they result in single event upsets and silent data corruption, leading to application crash and wrong output. Soft errors will continue to be an important issue due to shrinking feature sizes, reduced voltages, and increasing clock frequencies.

The register file is an important unit where errors need to be avoided since it is accessed very frequently and an error in registers can quickly propagate to the other parts of the processor. It is also shown in the previous research that the register file is especially vulnerable to soft errors since they lead to application failure [14]. Since the register file access time is on the critical path and it consumes 16% of the processor power [5], designers need fault tolerance methods that have low delay and power overheads.

Many researchers and microprocessor designers studied soft errors that occur in the microprocessor register file and processor logic. IBM G5 uses ECC to protect the register file [12] but it is reported that ECC consumes 10 times the energy than a non-ECC register access [11]. ECC requires roughly 4 times more power than the single-bit parity calculation in the common, error free case for 64-bit data. Similarly, the encoder/decoder of ECC is also 4 times bigger than that of the parity. The Itanium processor uses a parity bit to detect register file errors, and terminates the application if a register parity error occurs [6][4]. Triple-modular redundancy (TMR) is a traditional redundancy mechanism in which the hardware is replicated three times and the majority voting is used to avoid an invalid value generation [10]. As TMR is a very expensive method of the soft error protection in terms of area and power, it cannot be used for register files. Compiler directed methods that duplicate and re-schedule instructions for higher register reliability, like proposed in [6], increases the code size extensively. Using partially unused registers to duplicate the information is also researched as a way to avoid the soft-errors on the register file [3]. A better approach to protect the register file form soft-errors is replicating register values into unused registers [8]. Still, this method needs to create additional writes to create copies.

In this paper, we propose a simple and low-overhead scheme called "Recovery through Existing Copy" or REC, to design a soft-error resilient register file. While other fault tolerance solutions increase the area, delay, or power overhead to create redundancy, our solution exploits the already available redundancy inside the register file and relies on the fact that many of the register values have a replica inside the register file. To correct soft errors, we propose to use the existing redundant copies in the physical register file by keeping track of physical registers that have the same value. We observed that many of these copies can be detected at the ALU stage by only comparing input registers and the result value. REC provides error correction capability with the encoding overhead comparable to that of a single-bit parity scheme and area overhead comparable to that of the ECC. Note that REC reduces –does not eradicate- soft errors by detection and recovery. However, REC scheme can be a viable alternative for designers who try to achieve acceptable MTTF targets with minimal power and delay overhead.

The REC solution can be used in conjunction with the schemes that try to create the redundancy inside the register file by creating copies of the values that were not previously present. In order to further improve the effectiveness of REC, we propose replicating allocated registers over unused registers to increase the number of copies as explained in [8]. By using this improvement we increased the ratio of registers that have a redundant copy inside the physical register file.

The rest of the paper is organized as follows. Section 2 presents the motivation of the study. Section 3 explains the proposed design. Section 4 presents the evaluation setup. Section 5 presents the results and finally Section 6 concludes.

2 Motivation

Our design was motivated by the key observation that on the

Manuscript submitted: 13-Feb-2015. Manuscript accepted: 16-May-2015
Final manuscript received: 17-May-2015

average 66.1% of valid registers already have redundant copies in the physical register file on each cycle. Fig. 1 shows the average ratio of registers whose value already exists in another register. We found that 31.4% of these duplicates are due to the result of the instruction being the same as one of the source registers, i.e. instruction reads a value from a register, and writes the same value back to another register. This high ratio of redundant copies is caused by instructions such as register move operations and addition or subtraction with zero. Remaining duplicates are due to push-pop operations (where after increment and decrement operations stack pointer is renamed to different registers and clone values are created) and array processing instructions that load the same memory location to different physical registers.

Some micro code examples from *astar*, *specrand*, and *povray* benchmark traces are shown in Table 1. After renaming is completed, architectural registers are mapped to physical registers (marked in gray with "pr" prefix). The final state of the physical register file after register writeback is shown next to each code block. First block shows how redundant copies are created by `mov` instructions. Second code portion shows two load microops (in close proximity) loading a value from the same memory location to R6 register, which is renamed to different physical registers, creating the replica of the same value. The last code portion of Table 1 shows stack operations from *specrand* benchmark. As the RSP gets updated, it is renamed to different physical registers, creating replicas of stack pointer inside the register file.

3 Proposed Design

Detecting the identical values inside the register file cannot be easily achieved as the identical values can be present in any register, and any instruction that updates a register has to check the contents of all registers to find an exact replica. In order to make this replica detection operation feasible, we propose to exploit the instructions that have their result value equal to at least one of the source operands. Since many of the replicas are created by instructions that do not make a modification to their source operands, replica values can be detected at the execution stage. When there is no difference between a source operand and the result, the location of the replicas is stored along with the generated value.

In order to detect the replica values, we propose to add a

TABLE 1
Sample Code Fragments Creating Redundant Copies

```
MOVING (from Astar)
 mov rsp(pr151),rbp(pr70)
 ...
 mov rsi(pr92),0xce60
 ...
 mov rsi(pr43),0xce60
```

43	0xce60
70	0x613380
92	0xce60
151	0x613380

```
LOADING FROM SAME MEMORY LOCATION (from Specrand)
 ld  r6(pr107),[r14,24]
 ...
 ld  r6(pr41),[r14,24]
```

41	0xffff8800015fb000
107	0xffff8800015fb000

```
PUSH-POP STACK OPERATIONS (from Povray)
 sub rsp(pr154),rsp(pr39),8
 ...
 sub rsp(pr91),rsp(pr154),8
 ...
 sub rsp(pr67),rsp(pr91),32
 ...
 ...
 add rsp(pr30),rsp(pr67),32
 ...
 add rsp(pr147),rsp(pr30),8
 ...
 add rsp(pr57),rsp(pr147),8
```

30	0xffffffff814c1f10
39	0xffffffff814c1f10
57	0xffffffff814c1f08
91	0xffffffff814c1f00
147	0xffffffff814c1f00
154	0xffffffff814c1f08

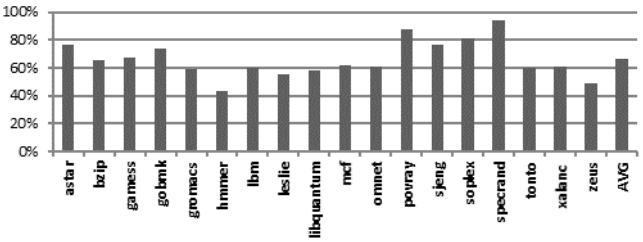

Fig. 1. Average ratio of valid registers having a redundant copy

comparison logic to check the equivalency of both sources (assuming instructions with 2 source operands) with the destination value. When the compared values are found to be the same and the registers' names are not equal (which is usually the case in out of order pipelines due to the register renaming), each copy logs the other register's name in an added field named "Copy Register Name" (CRN) as shown in Fig. 2. Only one register is allowed to assign another register as a replica and if an equal-valued source operand is already used as a replica of another register, it is not assigned as a replica of a destination register.

We propose to add a bit vector that contains one bit for each register, called "Copy Register Presence Bit Vector" (CRPBV) which is accessed when a fault occurs to see if the replica is still valid, before the replica is accessed. When two registers are found to be copies of each other, CRPBV is updated. If a register that is previously identified to be holding a replica value is overwritten, the bits corresponding to this register and its replica are cleared in CRPBV. This way, it is possible to identify the void replicas that are nullified by a subsequent overwrite operation and can no longer be used to recover a soft error on the previous replica. Destination register's CRN has to be read before it is overwritten in order to index CRPBV and clear the corresponding bit of its replica.

The added fields of CRN and CRPBV are kept in a separate storage since accesses to these fields are independent of the register file access and occur only when a copy or a fault is detected. Therefore, we do not add more ports to the actual data storage space, but the number of write ports of the added field is twice the number of write ports of the data storage. The number of read ports of the added field is the same as that of the data storage. Thus, the total number of ports in the data storage is unchanged and the number of ports in the added field is slightly higher than the data storage. Accesses to this field do not increase the delay overhead since it is much smaller than the register file and updates to it is done in parallel to the register writeback.

As stated by the previous studies, single-bit errors are the most common [8] and a parity check would detect most of

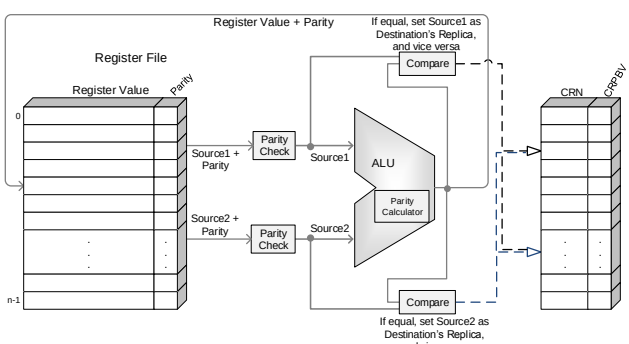

Fig. 2. Register file architecture for soft error protection

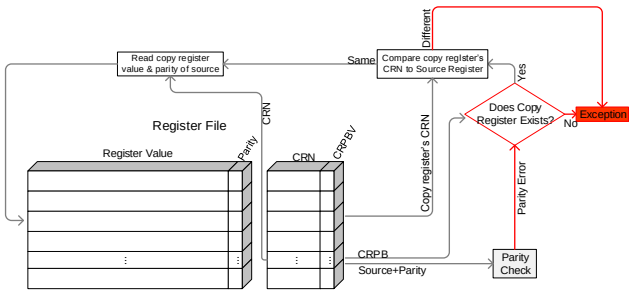

Fig. 3. Parity check and recovery flow diagram on register file architecture.

the faults. We add a parity bit field to each physical register. During register read, a parity check is performed, and during register write, parity is calculated at the ALU stage. Using REC, it is possible to extend the coverage of error detection to error correction by using the parity bits, as we now have a replica of the value and we know if the replica is valid or not. Error detection through the use of a parity bit is beneficial even when a replica does not exist since a solution can be found at the application level by creating an exception.

The parity check and recovery flow is shown in Fig. 3. The parity check is done when a source register is read. In REC, the parity of the source register is read along with its value. If the parity check shows that there is an error in the register value, then the presence of a copy register is checked by performing a lookup on the corresponding bit of the register in the CRPBV. If the source register does not have a copy register in the register file, then an exception occurs and the running application must be informed. Otherwise, the copy register, its parity, and CRN field are read to recover from the soft error. The copy register's parity is also checked again to determine the copy register's soft error. The reason we also read the CRN field of the copy register is to make sure that if a soft error occurs at the added fields (namely, CRN and Copy Register Presence Bit - CRPB), we can detect the error by comparing the CRN field of the replica with the source register itself. If they are different, due to an error at the added fields of the source register, the error is detected.

In order to extend the coverage of REC, we suggest an improvement to the design by exploiting the idle registers and creating a redundant copy of the register values as it was proposed by the study of Memik et al. [8]. We exploit the unused (free) registers to write the result of a register writing operation along with the destination register. To accomplish this we added a mechanism to the processor to allocate two of the free physical registers at the rename stage; one of them is used as the destination register, as usual, and the other is used as the copy register in a new state called "duplicate". Registers in the "duplicate" state are treated as "free" registers at the rename stage to allocate the destination register for avoiding any performance degradation in the common case of no error. But for the allocation of another duplicate register, the "duplicate" state is treated as used (valid). When a duplicate register is allocated along with the destination register, the result of the instruction is written to both of the registers, each register writes the other register's name to their CRN field, and corresponding bits of the CRPBV are set. After this point, the design works just the same in terms of the parity check and recovery. We call this extension as "register duplication mechanism" or RDM.

4 Simulation Environment

The architectural simulations are performed using a modified version of MARSSx86 [9], a full-system microarchitectural simulator which uses the x86 ISA and creates RISC-like microops to execute in the pipeline. We used 18 benchmarks from the SPEC CPU2006 [15] using the reference input sets compiled for the x86-64 architecture with the optimization level O3. We run the compiled benchmarks on a 64-bit out of order processor with the integer register file size of 160. The register file size is selected similar to the Intel Itanium Poulson, which is designed for mission critical servers [13]. We simulated each benchmark for 100 million user instruction commits after fast-forwarding for 1 billion instructions. The power simulations are performed using the Cadence design tools employing UMC 90nm technology node. A V_{DD} of 1V is used under 27°C ambient temperature.

5 Results and Discussions

We can use two metrics that can demonstrate how successful our schemes are in terms of error recovery; coverage and reliable read rate (as it was defined in [8]). Coverage is defined as the ratio of registers that have replicas over all registers, while reliable read rate is defined as the ratio of registers that are read and have replicas over all registers that are read. The reliable read rates of the detected copies in REC are shown in Fig. 4 on the left bars. We also show the potential reliable read rate if all replicas could be detected, on the right columns of the figure. The reliable read rate of detected copies is 20.4% on the average, reaching up to 36% for individual benchmarks, while the potential reliable read rate is 55.6% on the average, reaching more than 70% for some benchmarks.

The results of the combination of REC and RDM are shown in Fig. 5, where the potential coverage increased to 69.4% as shown in the rightmost columns. The potential reliable read rate also reaches 69.1% (middle column in Fig. 5) showing a significant improvement. Since our proposed scheme also detects the duplicated values created by RDM, the leftmost bars for each benchmark in Fig. 5 shows both the percentage of the detected copies that already exist in the register file and the percentage of the duplicated registers created by RDM. The stacked column on the left shows these constituents of the detected copies, upper portion being the reliable read rate due to RDM. Reliable read rate due to all detected copies are 44.1% of all registers.

We injected faults to the register file to see the success rate of the proposed schemes. After fast-forwarding 1 billion instructions, we injected one fault to a random location in the register file at a random time and we trace the fault for 10 million instructions to see if the fault is recovered. In total, we inject 200 faults per benchmark and we make one fault

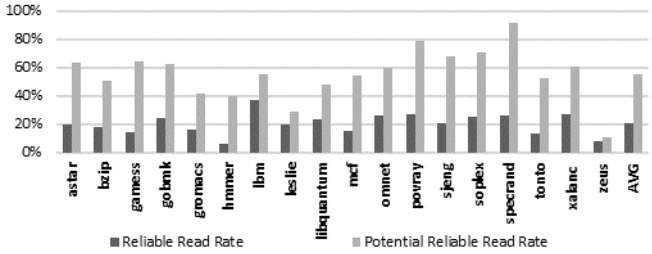

Fig. 4. Reliable read rate using REC.

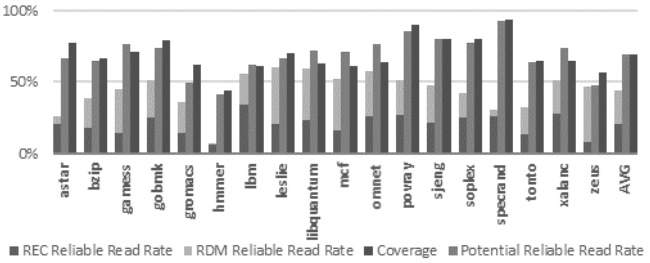

Fig. 5. Results for the REC with RDM.

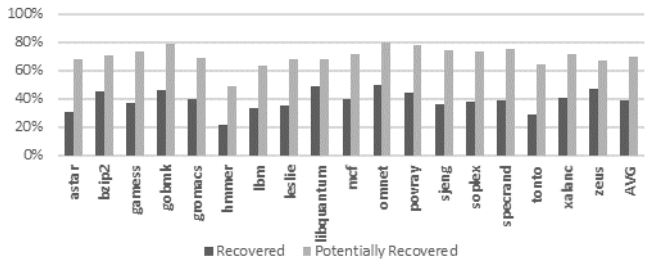

Fig. 6. Error recovery results for randomized error injection.

injection in one execution. This fault injection methodology is similar to prior analyses [16][7]. The left column in Fig. 6 shows the error recovery rates with the proposed architecture while the right column shows the percentage if all of the copies in the processor would be detected. The proposed design has 39.0% error recovery on the average, while the potential recovery rate is 70.1%. It was reported that SEU rate per microprocessor for 40nm technology is 0.94 FITs/Kb [2]. Therefore, the 160x64 register file we evaluated has 12.3 years of MTTF, while using our proposed method the register file's MTTF would increase to 20.2 years.

Finally we simulated the power consumption of a conventional unprotected architecture to compare it to our schemes. The extra power consumption of the proposed architecture consists of the power of reading destination register's CRPBV and CRN, writing to the bit vector field of its replica, parity calculation and checking, and the comparison logic. On the average the extra power dissipation of REC was only 2.8% without RDM. When we include RDM as explained in [8], the extra power consumption becomes 18.9%. This power dissipation overhead is very small when compared to the overhead introduced by the use of ECC which is reported to be as high as an order of magnitude by the industry [11].

Some contemporary processors use a technique called "Move Elimination" to avoid creating replicas in the register file for reducing power dissipation or improving performance [1]. This technique, when used with our scheme, will reduce the soft error correction coverage and has a tradeoff between power dissipation and reliability. We leave the dynamic switching between the Move Elimination and the proposed scheme for better performance/power/reliability tradeoffs for future work.

6 Conclusion and Future Work

In this paper we present Recovery through Existing Copy (REC) design to exploit already available replicas in the register file to make the register file less vulnerable to soft errors. For every result-producing instruction, source values and the result are compared to detect if the instruction is creating a replica value in the register file. We use parity protection on the main value and hold a pointer to its replica for error correction. We also showed that register duplication mechanism (RDM) as explained in [8] can be used in conjunction with the proposed scheme for achieving better error coverage. When used in conjunction with [8], our scheme corrects 39.0% of errors, with the reliable read rate of 44.1%. The power dissipation overhead of REC is 2.8% but it goes up to 18.9% if we extend the coverage by using RDM.

Our proposed solution provides error correction capability with the encoding complexity comparable to that of a single-bit parity scheme while the area overhead comparable to that of the ECC. Note that we provide multi-bit error correction coverage when compared to ECC with less hardware overhead although we provide this coverage for only the values that have a redundant copy in the register file. REC is orthogonal to and can be used in conjunction with the lower level approaches to reduce soft error rate (SER) such as gate sizing [17]. The methods proposed in this paper can only exploit 31.4% of the replicas existing in the register file. If full replica potential existing in the register file is used, it is possible to achieve 70.1% error recovery rate with a power consumption of 21.0%. Improving the coverage by exploiting the existing replicas more efficiently is left for future work.

Acknowledgment

This work was supported in part by TUBITAK under Grant 112E004. The work is in the framework of COST Action 1103

References

[1] S. Battle, A. D. Hilton, M. Hempstead, and A. Roth, "Flexible register management using reference counting," in HPCA, 2012.

[2] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in IEEE IRPS, 2011.

[3] O. Ergin, O. S. Unsal, X. Vera, and A. Gonzales, "Exploiting narrow balues for soft error tolerance," IEEE CAL, vol. 5, no. 2, 2006.

[4] E.S. Fetzer, D. Dahle, Little C., and K. Safford, "The parity protected, multithreaded register files on the 90-nm itanium microprocessor," IEEE JSSC, vol. 41, no. 1, pp. 246-255, 2006.

[5] D. R. Gonzales, "Micro-RISC architecture for the wireless market," IEEE Micro, vol. 19, no. 4, pp. 30 - 37, 1999.

[6] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M. Irwin, "Compiler-directed instruction duplication for soft error detection," in DATE, 2005.

[7] M. L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in HPCA, 2009.

[8] G. Memik, M. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors," in DATE, 2005.

[9] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A Full System Simulator for Multicore x86 CPUs," in DAC'11, 2011.

[10] W. Peterson, "Error-correcting codes". Cambridge: The MIT Press, 1980.

[11] R. Phelan, "Addressing soft errors in ARM core-based SoC," ARM White Paper, 2003.

[12] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in ISCA-2000, pp. 25-36.

[13] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, T. Grutkowski, "A 32 nm, 3.1 billion transistor, 12 wide issue Itanium Processor for Mission-Critical Servers," IEEE JSSC, vol. 47, pp. 177-193, 2012.

[14] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," IEEE Micro, vol. 25, no. 6, pp. 30-39, 2005.

[15] Standard Performance Evaluation Corporation, SPEC Benchmarks., 2006. [Online]. http://www.spec.org

[16] G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero, "FIMSIM: A fault injection infrastructure for microarchitectural simulators," in ICCD, 2011.

[17] Q. Zhou and K. Mohanram, "Gate sizing to radiation harden combinational logic," IEEE TCAD, vol. 25, no. 1, pp. 155-166, 2006.