Non-Uniform Fault Tolerance

Jonathan Chang George A. Reis Neil Vachharajani Ram Rangan David I. August
Departments of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08544
{jcone,gareis,nvachhar,ram,august}@princeton.edu

Abstract

As devices become more susceptible to transient faults that can affect program correctness, processor designers will increasingly compensate by adding hardware or software redundancy. Proposed redundancy techniques and those currently in use are generally applied uniformly to a structure despite non-uniformity in the way errors within the structure manifest themselves in programs. This uniform protection leads to inefficiency in terms of performance, power, and area. Using case studies involving the register file, this paper motivates an alternative Non-Uniform Fault Tolerance *approach which improves reliability over uniform approaches by spending the redundancy budget on those areas most susceptible.*

1 Introduction

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [1, 8, 13], rendering processors that use them more susceptible to *transient faults*. Transient faults, also known as *soft errors*, are intermittent faults caused by external events, such as energetic particles striking the chip. They do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values.

Incorrect execution in high-availability and real-time applications can potentially result in serious damage and thus these systems have the highest reliability requirements. These systems will often resort to exhaustive hardware redundancy to ensure maximum reliability. While several fault tolerance solutions have been proposed for high-end systems [4, 14, 18], the high hardware costs of these solutions make them less than ideal for the desktop and embedded computing markets.

The high cost of these techniques, in terms of performance, area, and power, is due in large part to the breadth with which they are applied. Existing techniques often resort to duplicating functional units, processor pipelines or even entire processor cores. These techniques are *uniform*; they protect all of the data within their spheres of protection uniformly.

This uniform approach leads to a prodigal use of system resources. As described later, the effects of soft errors are, in fact, *non-uniform* across several dimensions for registers — value, bit number, and register number. Since the effects are not uniform as existing techniques anticipate, existing techniques are inefficiently applying their resources. To take advantage of this non-uniformity, this paper proposes a *Non-Uniform Fault-Tolerance* approach, which targets the level of protection to where it is most necessary.

The notion of non-uniformity is already well-entrenched in various aspects of computer architecture. For example, caches take advantage of the fact that memory accesses do not follow a uniformly random distribution. Instead, they exhibit striking temporal and spacial locality which has been exploited to boost performance. Branch prediction is another case in which heavy biasing leads to significant speedups. However, non-uniformity has not yet been exploited to the realm of reliability.

To apply non-uniform fault tolerance, a designer begins by experimentally observing and quantitatively measuring the phenomenon of non-uniformity in faults. Once the nature of non-uniformity in faults is understood, the designer identifies those state elements which are most likely to corrupt program output and analyze why they cause user-visible errors. Finally, with these results, the designer can appropriately adapt existing techniques for non-uniformity in order to target only the most important elements. Such an approach allows designers to make the smallest set of modifications to their processor and incur the smallest penalties while still meeting their reliability goals. A non-uniform fault tolerance techniques can offer nearly perfect protection while incurring very little cost when compared against the original techniques. One such technique is able to effectively eliminate all output-corrupting faults due to single bit errors to the integer register file, while only requiring minimal changes to a modern processor.

The rest of the paper is organized to illustrate the non-uniform fault tolerance approach using case studies on the register file. Section 2 first provides background information for the transient fault detection discussion. Section 3 describes the experimental setup used in the exploration of several dimensions of non-uniformity in the register file.

Sections 4-6 evaluates and exploits non-uniformity across individual registers, bit positions within a register, and register values. The paper concludes with Section 7.

2 Preliminaries

This paper assumes the *Single Event Upset* fault model typically used in related work [9, 7, 12]. This model assumes that exactly one bit is flipped exactly once during a program's execution and is used because the underlying physical causes of soft errors make double-bit and multi-bit errors orders of magnitude less likely than single-bit errors. In this model, any bit in the system at any given execution point can be classified as one of the following [7]:

ACE These bits are required for *Architecturally Correct Execution* (ACE). A transient fault affecting an ACE bit will cause the program to execute incorrectly.

unACE These bits are not required for ACE. A transient fault affecting an unACE bit will not affect the program's execution. For example, unACE bits occur in state elements that hold dynamically dead information, logically masked values, or control flows that are Y-branches [16]

Fault tolerance mechanisms add redundancy to reduce the total number of ACE bits by converting them to unACE bits *or* reduce output corruption by detecting faults to ACE bits. Unfortunately, most fault tolerance mechanisms do not provide perfect protection and, consequently, ACE bits remain and not all faults to these bits will be caught. When an ACE bit becomes corrupted and no fault protection mechanism detects the error, the program will exit abnormally (e.g. with a segmentation fault) or *Silent Data Corruption* (SDC) will occur. When SDC occurs, some program output is incorrect and the user is given no indication that an error has occurred. On the other hand, abnormal program termination (ABTERM) does not necessarily imply that data has been corrupted and additionally the user is notified of abnormal termination. While both SDC and abnormal termination are undesirable, designers attempt to almost entirely eliminate SDC to prevent silent corruption. Consequently, designers, to meet the reliability demands of their clients, typically have a target SDC rate of one fault per 1000 years, while they have a target ABTERM rate of one fault per 10 years [2]. The differentiation of these two classes of events by two orders of magnitude implies that SDC poses a far more signficant challenge to designers than ABTERM.

Mean Time To Failure (MTTF) is a traditional metric for comparing fault-tolerance systems. This metric measures the amount of time between system failures and depends on, among other things, specific manufacturing and environmental parameters. An alternative metric, *Architectural Vulnerability Factor* (AVF) [7], has been proposed that eliminates dependence on these quantities. AVF is defined as follows:

$$\text{AVF} = \frac{\text{number of ACE bits in the structure}}{\text{total number of bits in the structure}}$$

The two quantities are closely related; MTTF is inversely proportional to AVF where the scaling factor is determined by manufacturing and environmental parameters. Note, that AVF can be partitioned into SDC and ABTERM AVF by considering only SDC or ABTERM bits (bits which, if flipped, would result in an SDC or abnormal termination respectively) in the numerator of the expression.

Partitioning the space of existing techniques allows for a better understanding of how to modify each of them to take advantage of non-uniformity. The space of fault-tolerance techniques can be broadly described as operating at either the *Logic Level* or the *Architectural Level*. Logic level techniques operate at the level of individual circuits or state elements and are applied where fine-grained, local protection are needed. Logic level include techniques such as functional unit duplication [14] and ECC. Architectural level techniques operate at much coarser granularities. These operate by using redundant threading [3, 6, 10], software-level instruction duplication [11], hardware-level instruction duplication [9], processor-core duplication [18], or a hybrid combination of the aforementioned schemes [12].

3 Experimental Setup

This paper investigates AVF non-uniformity in the integer register file because previous studies [12, 17] have shown that the integer register file is one of the largest contributor to the processor core's overall SDC. While integer register file is the subject of this work, the conclusions about non-uniform fault tolerance certainly apply elsewhere in a modern processor core.

We evaluated the AVF of the integer register file for a variety of benchmarks culled from SPEC CPUINT2000, SPEC CPUINT95, and MediaBench [5] suites. The binaries were compiled using `gcc -O2` version 3.4.1 targeting PPC970.

Faults were injected by first selecting a random instruction uniformly distributed across the program's dynamic instruction stream. At that point in the program, a random bit from a random integer register is flipped and the program is then allowed to run to completion. 600 runs were executed for each benchmark and the average AVF across all benchmarks has less than 1% error with a 95% confidence interval.

4 Non-Uniformity Across Registers

One perspective from which to examine the distribution of faults is at the register level. Figure 1 shows the SDC of each of the 32 integer registers averaged across all of our benchmarks. Figure 2 similarly shows the corresponding ABTERM AVF.

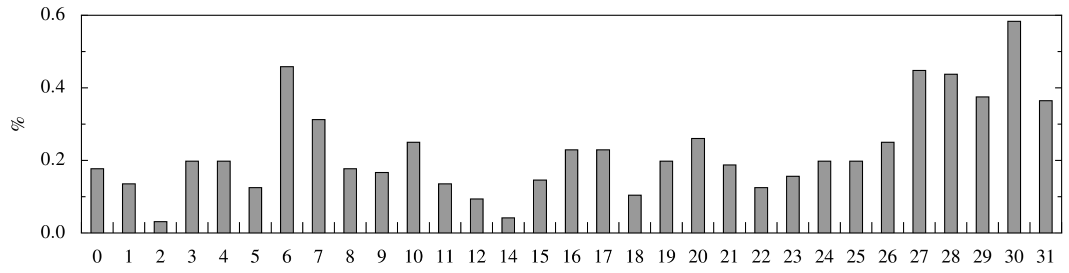

Figure 1. SDC AVF of the integer register file broken down by register.

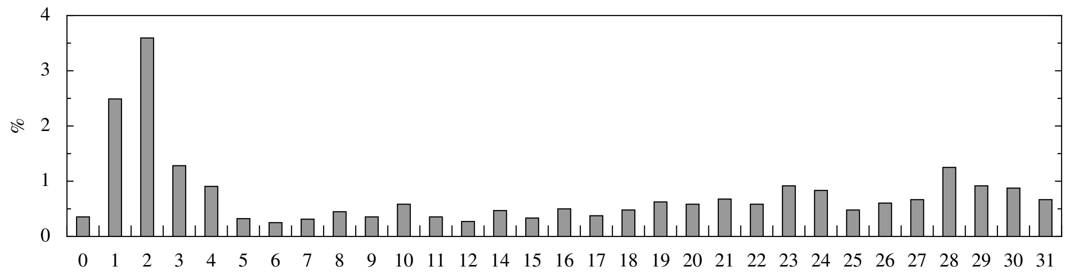

Figure 2. ABTERM AVF of the integer register file broken down by register.

4.1 Analysis

Figure 2 shows that the ABTERM AVF across registers is mostly uniform except for an extremely sharp peak which includes registers 1, 2, and to a lesser extent 3. In fact, these three registers account for 31.5% of overall ABTERN AVF. This suggests that providing extra protection to these registers will yield large dividends when trying to prevent ABTERM events.

The reason for the peak at these registers becomes evident when the PowerPC64 ABI [15] is considered. Register 1 is used as the stack pointer and thus is liable to cause a segmentation fault at one of the numerous stack references at the prologue and epilogue of each function. Register 2 is used as a Table of Contents (TOC) pointer. Code following the ABI specification rarely refers to data directly. Instead, the linker fills a table of contents with the address of each datum at link time so that code never needs to be modified in order to support proper linkage. When code wishes to access a data item, it loads the address of the item from a TOC entry which is a fixed offset from the TOC pointer, register 2. Thus, any fault to register 2 will likely result in the program's inability to properly access any datum. Finally, register 3 is used both as the first parameter register and the return value register. Since parameters and return values are often pointer values, faults to this register may also cause segfaults.

Figure 1 exhibits much more irregularity than Figure 2. Despite this fact, there are two regions where the SDC AVF is higher. One is the region centered around register 6, while another one is centered register 30. Again, this can be explained by examining the ABI. The parameter registers (registers r3 through r10) are caller-saved. Thus, any function which calls another function must save its own parameters by spilling. Typically, these registers are spilled into the callee-saved registers (r14-r31) going backwards from r31. Thus, registers like r30 and r31 are almost always live. Furthermore, their live-ranges are typically very long because they span child function calls. This makes them susceptible to a higher SDC AVF because the longer a register is live, the more opportunity there is for it to be corrupted.

Register 6, being a caller-saved register, is volatile, and so many functions without any child function calls will use r6 as temporary storage for computation. It's particularly high SDC here is due mostly to two benchmarks — `adpcmdec` and `adpcmenc`. In both cases, r6 is used throughout a function which comprises the bulk of the execution time of the programs. Furthermore, output depends directly on every bit of this register, ensuring that the SDC of this register is essentially 100% for these two benchmarks. We observe that the code that manipulates and uses r6 can be easily modified so that only the least significant bit is ever live, suggesting that compiler transformations may be able to eventually manufacture non-uniformity to make codes more amenable to low-cost fault-tolerance techniques.

4.2 Exploiting Non-Uniformity Across Registers

The prior section has examined the non-uniformity of errors across registers, showing that certain registers contribute more to the overall SDC AVF of the system than others. It is therefore desirable to protect only those registers which are the top contributors.

Suppose, for example, that a system were able to offer single-error correction for half of the 32 registers. In this case, the designer should choose to protect the 16 registers

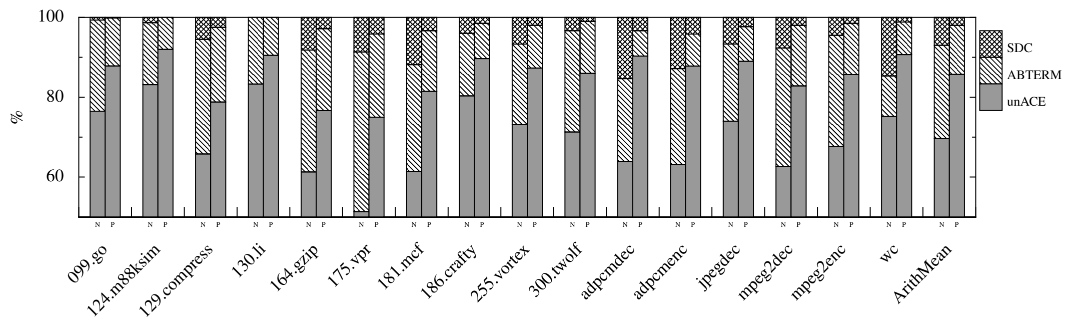

Figure 3. AVF of the integer register file after protecting the 16 registers with the highest SDC AVF (right) and the AVF without any protection (left).

with the highest SDC AVF. The hypothetical result of such a choice is given in Figure 3. The bars on the left in show the original, unprotected AVF. The bars on the right show the result with the protection scheme.

This scheme, while simpler and drastically less costly than full register file protection, is able to reduce the SDC AVF from 7% down to 2%. This corresponds to a normalized MTTF of 3.5x. Furthermore, the ABTERM AVF is reduced from 23.4% down to 12.3%.

5 Non-Uniformity Across Bits

Another perspective from which to view non-uniformity is at the bit level. Figure 4 shows the SDC AVF of each bit of the average 64-bit integer register, averaged across all benchmarks.

5.1 Analysis

Figure 4 has some clear peaks. This is most evident at bit 7. Bit 7 is the most significant bit for byte operations, and is therefore often duplicated via sign-extension operations, effectively transforming a 1-bit fault into a 57-bit fault.

Even more striking than the peaks, however, is the sharp decay of the SDC AVF as the bit number increases. This can be seen more clearly in Figure 5, the cumulative distribution of Figure 4. The curve flattens out rapidly after the midway point. In fact, this graph indicates that 87% of the SDC faults occur on the lower half of the register. 78% of the SDC faults occur on the lower 24 bits of the register and 63% on the lower 16 bits of the register.

Many computations, especially in C code, operate on data which are less than 64 bits. The `int` datatype, for example, is 32-bits. Therefore, faults on the upper-bits are likely to be logically masked by sign and zero extension operations. Furthermore, faults on the lower bits of a pointer value are more likely to yield another valid pointer while faults on the upper bit are not. Thus, for pointers, faults on the lower bits will lead to the reading or writing of incorrect data while faults on the upper bits will lead to segmentation faults.

The non-uniformity which dominates this graph suggests that techniques which only protect the lower bits of registers are likely to do almost as well as techniques which protect the entire register.

A slightly different picture emerges from the corresponding graphs for ABTERM AVF. These are shown in Figure 6 and Figure 7. These graphs indicate that a smaller number of the lower bits are ABTERM bits. In contrast to 63% of the total SDC AVF being contributed by the lower 16 bits, the ABTERM AVF for the lower 16 bits is only 18% of the total ABTERM AVF. This observation is further borne out by the CDF in Figure 7. The graph begins as a fairly flat linear curve, but around bit 20 an inflection occurs, after which the line trends upwards more sharply.

Again, we observe that faults to the upper bits are more likely to resteer pointers into invalid addresses. This inflection occurs at bit 20 because of the way in which pool memory allocation works in the system libraries and operating system used. To verify this theory, we wrote a small test program which would simply allocate 100 bytes via the `malloc` call and then successively access addresses starting from the beginning of the allocated area until a segmentation fault occurred. The resulting program produced a segmentation fault 2^{21} bytes from the start of the allocated area. This suggests that the distance from an average point in allocated heap memory to a segfaulting address is 2^{20}.

5.2 Exploiting Non-Uniformity Across Bits

As shown, the majority of SDC bits are concentrated in the lower 32 bits of registers. If a technique were to have single-error correction for the lower 32 bits of every register, while having no protection for the upper 32-bits, then it would have the reliability characteristics given in Figure 8.

The bars on the left in Figure 8 show the original, unprotected AVF. The bars on the right show the result with the hypothetical protection scheme. On average, the SDC plummets from 7.0% down to 0.8%. This corresponds to a normalized Mean Time To SDC Failure of 8.5x. Clearly, taking advantage of non-uniformity to intelligently apply fault-tolerance can yield large dividends. Meanwhile, this simple change also reduces the amount of ABTERM AVF from 23.4% down to 13.2%.

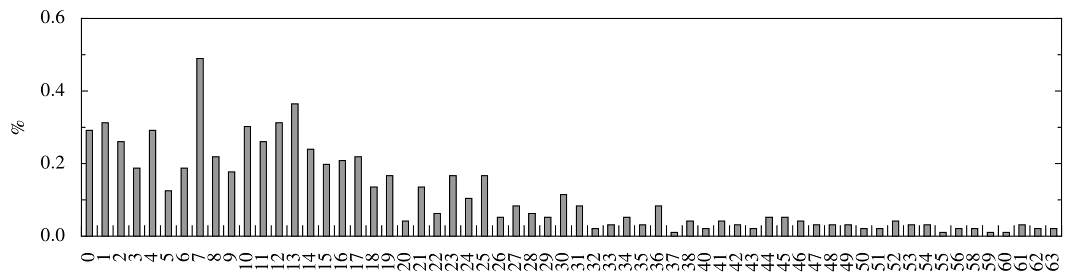

Figure 4. SDC AVF of the integer register file broken down by bit.

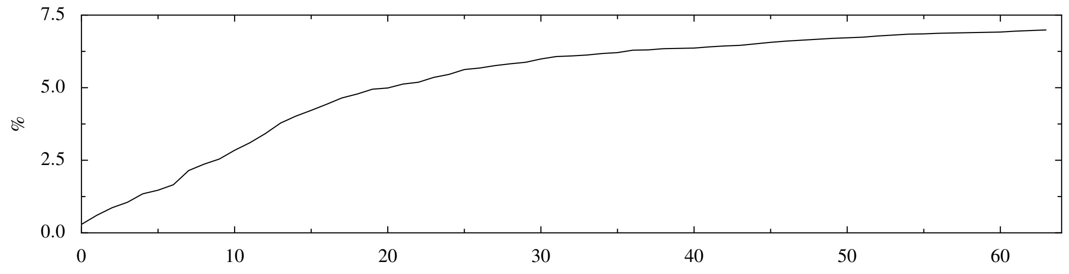

Figure 5. CDF of the SDC AVF of the integer register file.

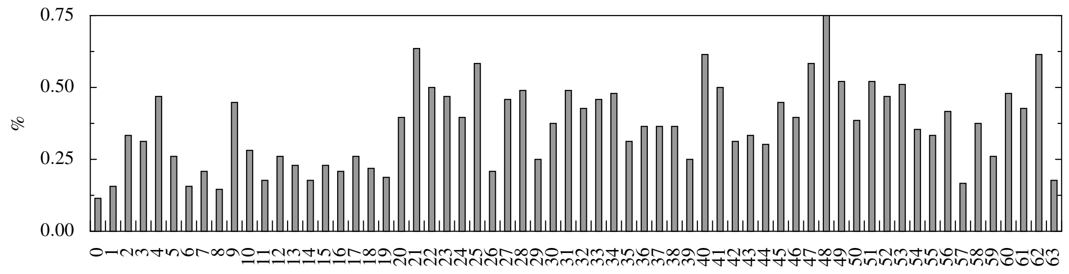

Figure 6. ABTERM AVF of the integer register file broken down by bit.

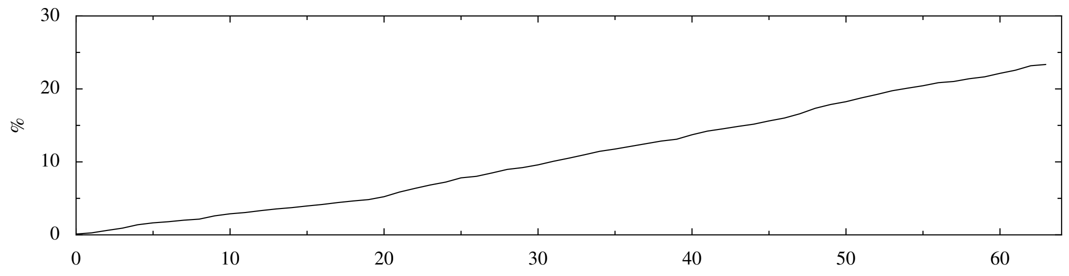

Figure 7. CDF of the ABTERM AVF of the integer register file.

Values	Frequency	SDC Frequency	ABTERM Frequency	Encoding
`0x00000000`	88.8%	91.7%	82.3%	`00`
`0x000001FF`	6.3%	0.0%	14.7%	`10`
`0x00000080`	2.2%	8.3%	3.0%	`11`
`0xFFFFFFFF`	1.4%	0.0%	0.0%	`01`
`0x90000000`	0.3%	0.0%	0.0%	`01`
Other	1.0%	0.0%	0.0%	`01`

Table 1. Original values of the upper 32 bits of the register.

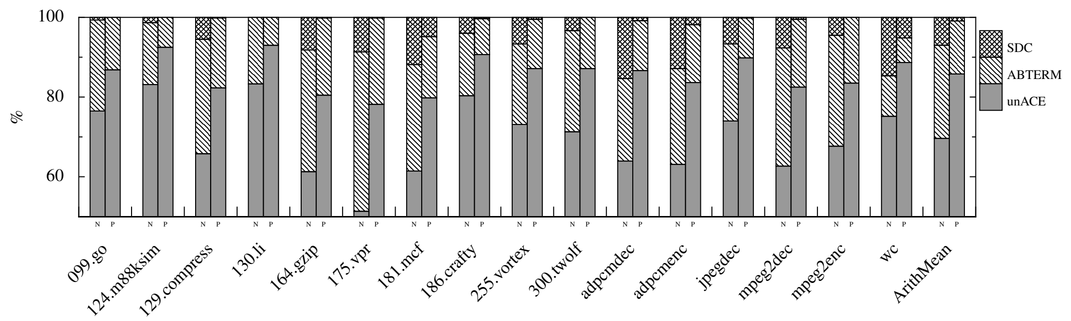

Figure 8. AVF of the integer register file after protecting the lower 32-bits of each register with a single-error-correcting scheme (right) and the AVF without any protection (left).

6 Non-Uniformity Across Values

The actual values held in the register file also follow a non-uniform pattern, as some values may be more prevalent than others. Because of this, the information complexity of the values in the registers may be far less on average than in the worst case. This means that stronger protection can be more easily applied.

For the first 100 runs of each benchmark, we recorded the original value contained in the register right before it was altered. The first and second columns of Table 1 give the breakdown of the upper 32 bits of these values. This can be considered to be a random sampling of the upper bits of the values in the register file. Clearly, the upper bits of the register usually only assumed one of a few values, leaning very heavily towards the value `0x0`. The top three entries on this table alone account for 97.3% of the values in the upper part of any register at any given time.

6.1 Analysis

The third column of Table 1 shows the breakdown of values in the upper 32 bits whenever flipping one of those bits led to SDC. In this case, the upper bits only ever assumed one of two values, namely `0x0` and `0x80`, again leaning heavily towards `0x0`.

A similar phenomenon occurs when one examines ABTERM bits in the fourth column of Table 1. Again, only a very small number of distinct values is ever assumed. And once again, `0x0` is by far the most prevalent value. A value of `0x1FF` in the upper 32 bits of a register would strongly imply that the register was a pointer into the stack segment since stack addresses always have `0x1FF` in their upper bits. By the same token, a value of `0x80` in the upper bits would imply that the register is a pointer to a heap address. A value of `0x0` in the upper bits may either refer to addresses in the code, data, or TOC segments. In this situation, a segmentation fault resulting from a fault to the upper bits of a register containing `0x0` in its upper bits may mean that the register contained an offset or some intermediate in the offset computation. Offsets are small and predominantly positive.

6.2 Exploiting Non-Uniformity Across Values

Whenever a fault to the upper 32 bits of a register yielded an SDC or ABTERM, Table 1 indicate that only three distinct values were ever taken. This means that a fully redundant copy of the upper 32 bits of the data in these cases can be encoded in only $\lceil \log 3 \rceil = 2$ bits. A suggested encoding is given in the last column of Table 1. The remaining encoding value can be used in the unlikely event that the upper bits do not correspond to any of the three predefined value. In this situation, the level of protection will be reduced.

Since every valid codeword (for the three observed cases) is now a hamming distance of more than two from any other valid codeword, this scheme provides complete single error correction for the upper 32-bits. When this technique is combined with the technique described earlier for protecting the lower 32 bits, yielding a single-error correction technique for registers, for which we observed perfect reliability, at a significantly lower cost than traditional techniques.

7 Conclusion

By investigating the integer register file of a PPC970 in detail, this paper shows that the distribution of AVF is non-uniform across bits, registers, and values. This information motivates modifications to existing fault-tolerance techniques to take advantage of this phenomenon.

By prioritizing protection to only those bits which need it the most, *Non-Uniform Fault-Tolerance* techniques can vastly reduce the cost of implementing an existing fault-tolerance technique while still maintaining its reliability characteristics. In one application, a non-uniform fault-tolerance technique was able to effectively eliminate all single-upset error induced SDC in the integer register file while incurring only minimal changes to a typical processor. This experience motivates more techniques to take full advantage of the inherent non-uniformity in the manifestation of errors.

References

[1] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.

[2] D. C. Bossen. CMOS soft errors and server design. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_07.1 – 121_07.6, April 2002.

[3] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109. ACM Press, 2003.

[4] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 216–226, May 1990.

[5] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[6] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110. IEEE Computer Society, 2002.

[7] 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 *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 29. IEEE Computer Society, 2003.

[8] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.

[9] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224. IEEE Computer Society, 2001.

[10] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM Press, 2000.

[11] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[13] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–399, June 2002.

[14] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[15] I. L. Taylor. 64-bit PowerPC ELF application binary interface supplement 1.7. http://www.linuxbase.org/spec/ELF/ppc64/PPC-elf64abi-1.7.pdf.

[16] N. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 56–67, September 2003.

[17] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 61–72, June 2004.

[18] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.