Compression-free Checksum-based Fault-Detection Schemes for Pipelined Processors

Bernhard Fechner, Jörg Keller, FernUniversität in Hagen,
Fakultät für Mathematik und Informatik, 58084 Hagen, Germany

Abstract

We propose a fault-detection scheme for pipelined, multithreaded processors. The scheme is based on checksums and improves on previous schemes in terms of fault coverage and detection latency by not using compression but storing complete checksums from several pipeline stages. We validate the scheme experimentally and derive checksum polynomials that lead to perfect fault coverage.

1 Introduction and Motivation

Currently, we see two trends in the microprocessor arena. On the one hand processors still get ever more complex, with the trend to multicore designs slowing down the complexity increase without stopping it. On the other hand, the ever increasing number of gates needed in a microprocessor can only be manufactured because of shrinking feature sizes. This trend has reached a point where particles are able to cause single event upsets (SEU) at ground level, something that previously has only been known to occur in space missions. Those soft-errors can be modeled like random bit-flips in registers [15]. The detection of such faults is necessary because they may lead to incorrect results which can have catastrophical consequences. A multitude of approaches has been published, ranging from triple-modular redundant systems on the heavy side to checksum approaches on the low overhead side. The proposed scheme belongs to the latter.

The rest of the paper is organized as follows: We will have a look at related work in Section 2. Section 3 reviews schemes to secure the checked pipelined execution from previous work. An extension is proposed and shown how a higher degree of fault coverage can be achieved. Section 5 concludes the paper with an outlook to ongoing and future work.

2 Related Work

Checksum approaches follow a common concept: checksums over the fault-free execution of a block of instructions are stored as references, and another checksum is computed when the instruction block is actually executed. The execution is considered faulty if the two checksums or signatures do not match. Checksum computation requires deliberate considerations for pipelined and multiscalar structures, and a plentitude of approaches is known: see e.g. [16][17][18][19][20][21] as well as [22] and the further references therein. Yet, all these approaches, including our own previous work, try to provide short checksums to keep overhead in space and computation time low. This is achieved by aggregating checksums. Typically, kinds of exclusive-or additions are used, which however are prone to mitigate faults that flip one bit in two places. For example, our own previous approach [22] achieves a fault-coverage of 83%. Furthermore, there is some latency before a fault is detected, because the aggregation over all stages has to take place before checksums can be compared.

In our current work, we investigate whether a further investment in chip area for fault detection can increase the fault coverage and reduce this latency. We achieve this by saving all checksum parts instead of aggregating them. Therefore, faults can be detected earlier without sacrificing fault coverage. Therefore we validate our approach with experiments and try to quantify the area fault coverage.

3 Pipeline Signatures

In this Section we review schemes to compute signatures from [22] on a micro-architectural level for the control and data path of a simple microprocessor, exploiting the SMT-based pipelined execution scheme [1]. The fault model assumes transient faults in the form of SEUs (Single Event Upsets). Furthermore, we assume one fault at a time in one component (pipeline stage). SEUs are modeled through bit-flips in latches or flip-flops. The signature computation involves cyclic redundancy check codes (CRCs) [2][3]. Code words are gained by polynomial division of the message polynomial $v(x) = \sum_{i=1}^{n} v_i x^i$ by the generator polynomial

$g(x) = \sum_{i=1}^{n} g_i x^i$. The message v(x) is composed out of the instruction stream and the contents of pipeline latches containing the control information and data for a stage. To clarify this, we exemplary start with the computation of a signature for a simple pipeline which is shown in Figure 1.

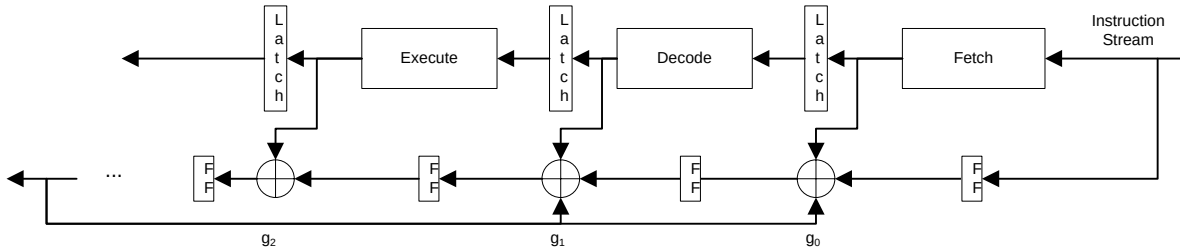

Figure 1. Signature computation

Here we already see a significant difference in comparison to the well-known CRCs. CRCs only use one input to the linear shift register, while we use all context from pipeline registers to complete the checksum. The implementation of error correcting/ detecting codes checking each register in a pipeline stage is possible, but was omitted due to the high additional implementation cost, power consumption and performance loss. A simple parity computation for each stage will affect performance, since we have to build the parity over all latches within a stage and thus have to face fan-in problems.

We can consider the contents of out-of-order pipeline stages to be a part of the checksum. This complicates the situation from Figure 1, since the dynamic multiple issuing from superscalar processors will lead to different parts of the control and data stream exiting the execution stage at different times. We can realize this part easily if we choose the generator polynomial in a way that no feedback affects the concerned stages. For flexibility we want to use any dynamic multiple issue policy. So we cannot use the scheme from Figure 1 without modification. The problem is to resolve the time dependency of instructions in the out-of-order stage. Thus, we have to choose an associative operation to build the checksum for this stage. Since XOR is associative, we will use it for a simple checksum computation in this part. We calculate two checksums separately, one for the out-of-order and one for the other in-order stages. At the time of a context switch, the two checksums will be combined. We switch the context on every latency-causing instruction (e.g. branches). Furthermore, branches will lead to the storage of the checksums. If the branch occurs in the second checksum, the existing checksums will be compared. If they are equal, no fault occurred. If not, an error will be signaled.

Figure 2 shows the checksum calculation including the out-of-order stage.

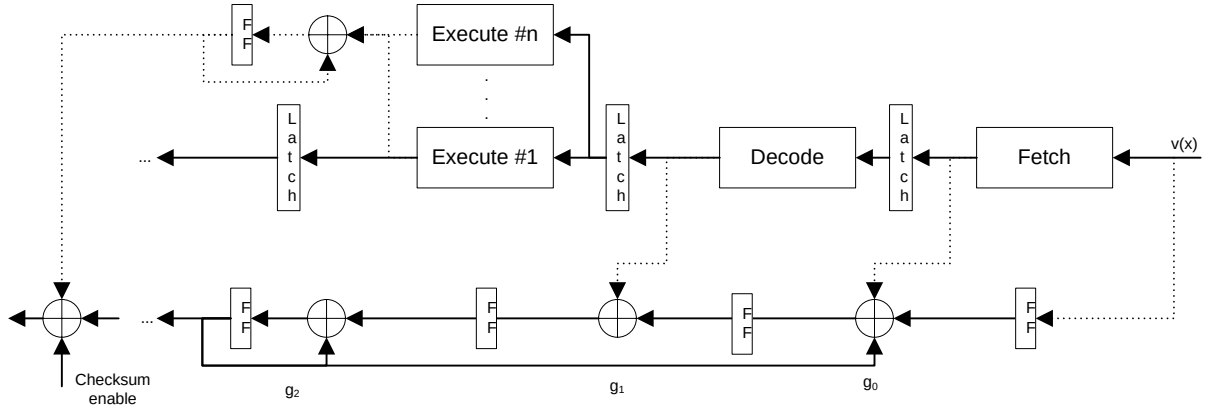

Figure 2. Out-of-order checksum calculation

To examine multithreaded execution, we assume two hardware threads. More hardware threads are possible, but were not considered, since two hardware threads are sufficient to detect a fault. Thus, to compare the calculated checksums in a multithreaded system, two context switches have to occur. A thread-ID tag in the pipeline helps to identify which instructions belong to which thread and to which checksum. Transient faults in the checksum selection by thread-IDs (TID) will lead to different checksums and to a detection of the fault. Figure 3 shows the checksum calculation for two threads.

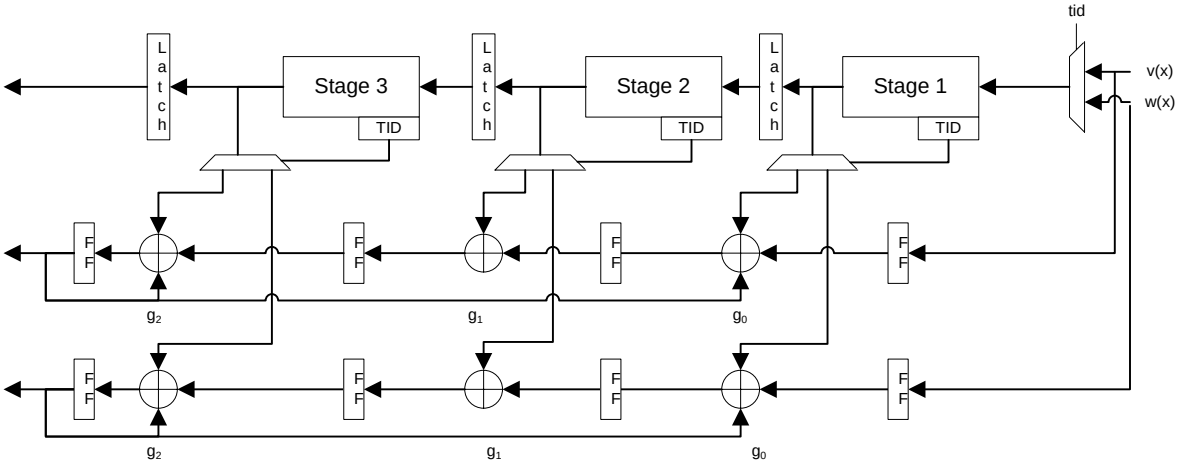

Figure 3. Checksum calculation for two threads

In Figure 3 we need additional time to completely output the checksum. The extension we propose is shown in Figure 4. The computed checksums for every stage will be directed in parallel to a checksum storage where they will be compared with previously calculated checksums.

4 Fault-coverage Analysis

We generated a random stream of 10^6 32 bit pseudo-instructions, which was used as an input for the modeled multithreaded processors, consisting of 5 and 8 pipeline stages, respectively. For simplicity, the contents of pipeline registers were not modified by different stages, the fetched instructions being propagated through the pipeline. The first experiment served to determine the polynomials with the best fault-coverage. Branches were created with probability $p_{branch} = 1/5$ (20%).

This probability was gained from SPEC95 benchmarks by using SimpleScalar [12] (see Table 1).

Table 1. Values for p_{branch} (%)

Benchmark	**Go**	**Ijpeg**	**Compress**
p_{branch} (%)	19.355	15.349	9.463
Benchmark	**Cc1**	**Apsi**	**Vortex**
p_{branch} (%)	24.251	22.546	22.931

We simulated transient faults in both instruction streams by flipping single, randomly chosen bits at random stages with a fault rate of 10^{-5} and an exponential distribution. No *warm-up phase* to fill the pipeline until the first checksum was calculated was considered. To determine minimum and maximum fault coverage values for different instruction mi-

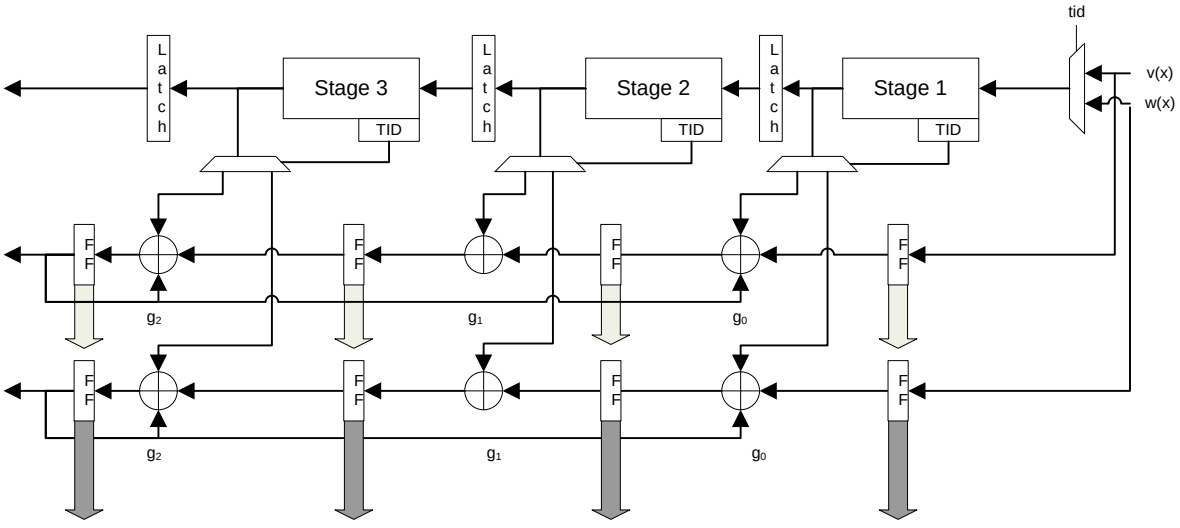

Figure 4. Parallel output of checksums

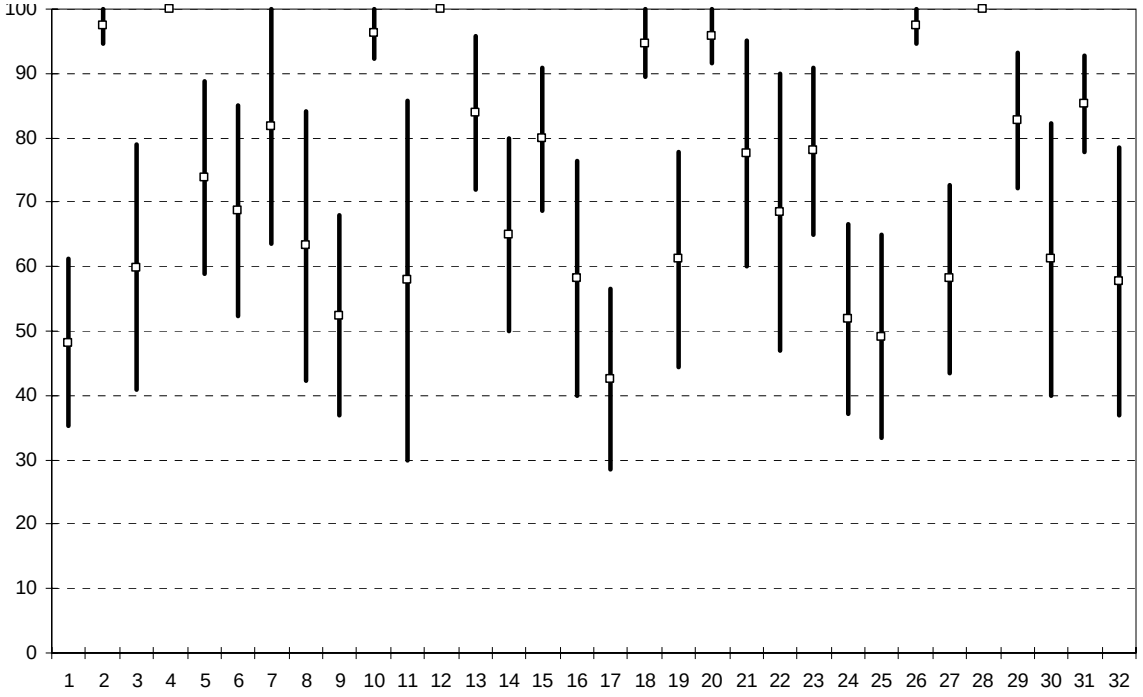

Figure 5. Fault coverage in % for a 5-stage pipeline (all polynomials)

xes, 10 fault injection runs were carried out. For a worst case study, we assumed that the pipeline will be flushed each time a fault is detected. Then the checksums and program counters will be rolled back to a previous sane state. This is the branch before the fault was detected. As first model we selected an in-order 5-stage pipeline with an internal control-path width of 32 bit from stage to stage. Multithreading is supported by using multiple instruction counters. Instructions will be fetched from an instruction stream until a branch occurs. Then the context is changed (*implicit* multithreading). If a branch is encountered in the second instruction stream both checksums will be compared. Figure 5 shows the results for a fault coverage analysis to get the polynomials with the best fault coverage. The x-axis shows the number of a specific polynomial, the y-axis the fault-coverage in percent. Please note that the labeling of the x-axis in Figure 5 and Figure 6 starts with 1 instead of 0.

We test all Boolean generator polynomials of degree ≤4, i.e. 32 different polynomials. This degree must be chosen as a maximum, because it is limited by the number of pipeline stages. We represent each polynomial by the decimal value of its coefficients seen as a binary number, i.e. the polynomial $g(x) = x^4 + x^3 + x + 1$ is represented by 27. The upper part of each column in the graph shows the maximum fault coverage, the lower part the minimum fault coverage. In each fault-injection run the instruction stream was re-generated.

We clearly see that the polynomials with 100% fault coverage are 3, 11, and 27 (offset -1). The polynomials with a fault coverage of more than 90% are 1, 9, 17, 19, and 25. Those are exactly the polynomials where the coefficient of x^0 is 1 and the coefficient of x^2 is 0. This indicates a relationship between the pipeline structure and promising generator polynomials. However, we are not aware of an analytic method to determine optimal polynomials. By using one of the mentioned polynomials, a perfect fault coverage was achieved, a 17% improvement in comparison to [22]. From CRCs it is known that the higher the degree of the polynomial, the higher the probability that a fault could be detected. In this context, we carried out the experiments again with the same parameters except the number of pipeline stages. Figure 6 clarifies this by showing the fault-coverage for an 8-stage pipeline. All polynomials of degree ≤7 were tested, but only the better ones are displayed. The symmetry in coverage supports the hypothesis mentioned.

Three polynomials for the 5-stage pipeline reached a perfect fault coverage in contrary to 117 for the 8-stage pipeline. It is plain to see that by applying a higher number of pipeline stages the fault-coverage increases. Figure 7 shows experimental results for a latency-based analysis of the enhanced scheme. The interesting question which inspired the analysis was how fast faults could be detected. We considered one polynomial (11) which showed perfect fault coverage for the 5-stage pipeline and the latency that results until the fault can be detected. ‘Time’ is a non-linear factor, since faults occur equally distributed in time. In the graph, we did not consider the computation of the latency over multiple fault injection runs. Thus, we have a high variance in the latencies. However, the arithmetic mean was 5,75

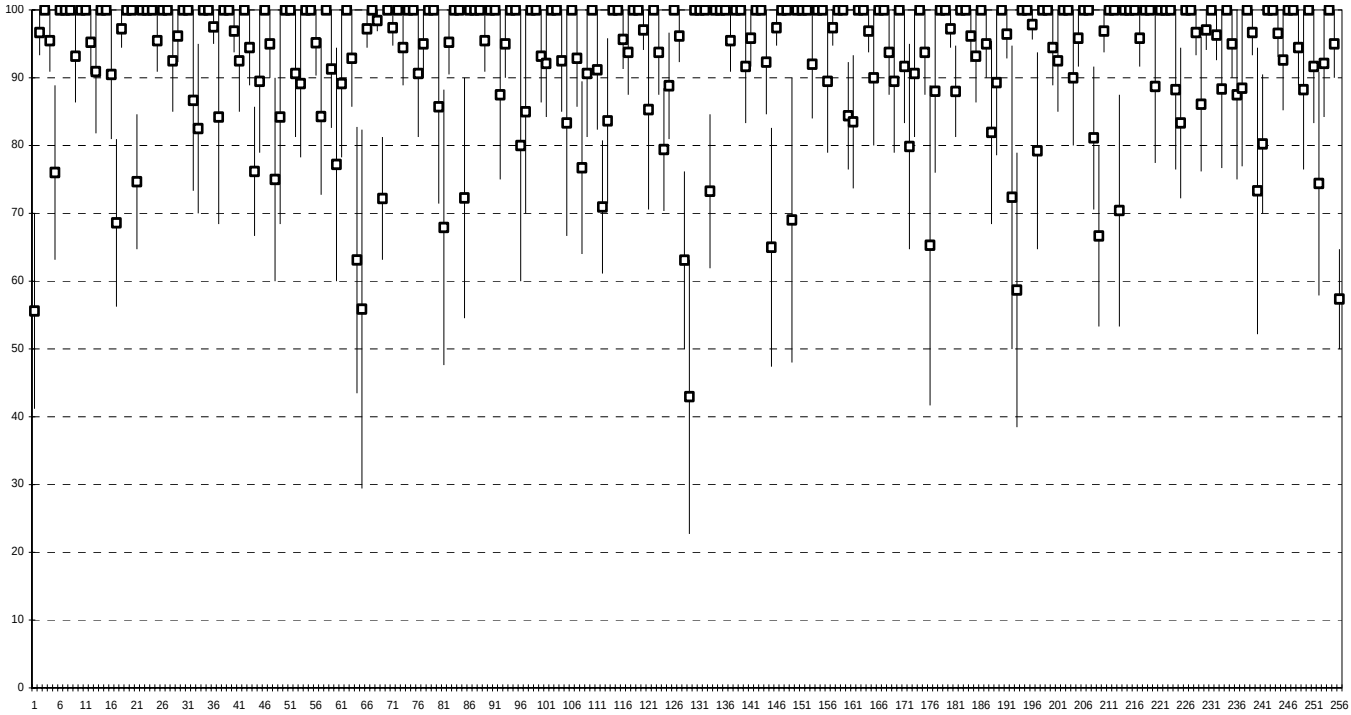

Figure 6. Fault coverage in % for an 8-stage pipeline (all polynomials)

cycles to detect a fault. No visible bar signals the immediate detection of a fault.

we have traded coverage against chip area. We have validated our approach by simulation and found

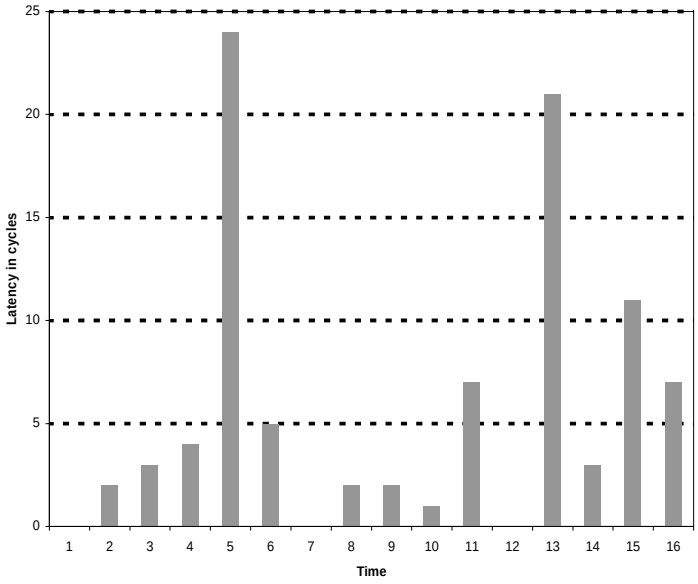

Figure 7. Latency in cycles to detect a fault (polynomial 11)

5 Conclusions and Work in Progress

We have presented a new approach to increase the fault coverage of checksum-based execution schemes for pipelined processors. To achieve this that we can reach perfect fault coverage, determined the polynomials and examined different pipeline lengths. For deeper pipelines, we can further increase the amount of polynomials to reach perfect fault coverage.

In our future work, we would like to investigate a compromise between the two extremes that we have followed here and in our previous work [22]: is it possible to partly compress the checksum with a function that is not as prone to mitigation of double flips as the exclusive or is? Also we would like

to extend the compression path that currently forms a cycle when the feedback is included, to a bi-cycle.

Furthermore, we will investigate how the fault-coverage will depend on the degree of instruction-level parallelism (ILP) since it is expected that it will negatively influence the fault-coverage. We will analyze how the fetch and issue policy will affect the fault-coverage. At the moment, we apply a block-multithreading strategy, switching the context on each branch. If we consider fine-grained multithreading, where we switch the context on a cycle-by-cycle basis, we expect the fault-coverage to increase that therefore to minimize the excepted loss in fault coverage for a higher degree of ILP.

References

[1] D. Tullsen, S. Eggers, and H. Levy, *Simultaneous Multithreading: Maximizing On-chip Parallelism*, 22nd Annual International Symposium on Computer Architecture, June 1995.

[2] S. Lin, D. Costello, *Error Control Coding*, Prentice-Hall, 1983.

[3] Peterson, W. & E. Weldon. *Error-Correcting Codes*, MIT Press, Second Edition, 1972.

[4] J.C. Smolens, B.T. Gold, J. Kim, B. Falsafi, J.C. Hoe, A. Nowatzyk: "Fingerprinting: bounding soft-error detection latency and bandwidth". ASPLOS 2004: 224-234.

[5] S.S. Yau, F.C. Chen. "An Approach to Concurrent Control Flow Checking". In IEEE Trans. Soft. Eng. SE-6(2) (March 1980): 126-137.

[6] M. Namjoo. "Techniques for Concurrent Testing of VLSI Processor Operation". In Proc. of the 12th Int'l. Symp. On Fault-Tolerant-Computing, IEEE Computer Society, Santa Monica, CA, June 1982, pp. 461-468.

[7] T. Sridhar, S.M. Thatte. "Concurrent Checking of Program Flow in VLSI Processors." In Digest of the 1982 Int'l. Test Conference, IEEE 1982, paper 9.2, pp. 191-199.

[8] J.P. Shen, M.A. Schuette. "On-Line Monitoring Using Signatured Instruction Streams", IEEE Proc. 13th Int'l. Test Conference, Oct. 1983, pp. 275-282.

[9] Richard W. Hamming. Error-detecting and error-correcting codes, Bell System Technical Journal 29(2):147-160, 1950.

[10] M.A. Schuette et al. "Experimental Evaluation of Two Concurrent Error Detection Schemes", In Proc. Of the 16th Int'l. Symp. On Fault-Tolerant Computing, Vienna, July 1986, pp. 138-143

[11] Karnik et al.: *Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes*, IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 2, April-June 2004.

[12] D.C. Burger and T.M. Austin. "The SimpleScalar Tool Set, Version 2.0", *Computer Architecture News*, 25 (3), pp. 13-25, June, 1997.

[13] S.M. Müller, W.J. Paul. *Computer Architecture. Complexity and Correctness*, Springer-Verlag, 2000.

[14] R. Baumann, *Silicon Amnesia: A Tutorial on Radiation Induced Soft Errors*. International Reliability Physics Symposium (IRPS), 2001.

[15] R.W. Wieler, Z. Zhang, R.D. McLeod, *Simulating static and dynamic faults in BIST structures with a FPGA based emulator*. In Proc. of IEEE International Workshop of Field-Programmable Logic and Application, pp. 240-250, 1994.

[16] Galla Thomas M., Sprachmann Michael, Steininger Andreas, Temple Christopher: Control Flow Monitoring for a Time-Triggered Communication Controller, May 6th-7th 1999, Proceedings of the 10th European Workshop on Dependable Computing (EWDC-10), Vienna, Austria.

[17] R. J. Andraka and J. L. Brady. A Low Complexity Method for Detecting Configuration Upset in SRAM Based FPGAs. MAPLD 2002, Proceedings of the 2002 Military and Aerospace Applications of Programmable Devices and Technologies Conference, Sept 10-12, 2002, Laurel, MD.

[18] Seongwoo Kim and Arun K. Somani. On-Line Integrity Monitoring of Microprocessor Control Logic. Proc. International Conference on Computer Design 2001,

[19] S. Kim and A. K. Somani, "SSD: An affordable fault-tolerant architecture for superscalar processors," in Proc. of IEEE 2001 Pacific Rim International Symposium on Dependable Computing (PRDC), December, 2001.

[20] Yung-Yuan Chen, Kun-Feng Chen. Incorporating signature-monitoring technique in VLIW processors. In Proc. 19th IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems (DFT 2004), pp. 395- 402, 2004.

[21] Wilken, K. D. and Kong, T. Concurrent Detection of Software and Hardware Data-Access Faults. IEEE Trans. Comput. 46, 4 (Apr. 1997), 412-424.

[22] B. Fechner. Analysis of Checksum-Based Execution Schemes for Pipelined Processors. Proc. 11th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems, Rhodes, Greece, April 2006.