Analysis of Checksum-Based Execution Schemes for Pipelined Processors

Bernhard Fechner

FernUniversität in Hagen, Department of Computer Science,
58084 Hagen, Germany
bernhard.fechner@fernuni-hagen.de

Abstract

The performance requirements for contemporary microprocessors are increasing as rapidly as their number of applications grows. By accelerating the clock, performance can be gained easily but only with high additional power consumption. The electrical potential between logic '0' and '1' is decreased as integration and clock rates grow, leading to a higher susceptibility for transient faults, caused e.g. by power fluctuations or Single Event Upsets (SEUs). We introduce a technique which is based on the well-known cyclic redundancy check codes (CRCs) to secure the pipelined execution of common microprocessors against transient faults. This is done by computing signatures over the control signals of each pipeline stage including dynamic out-of-order scheduling. To correctly compute the checksums, we resolve the time-dependency of instructions in the pipeline.

We will first discuss important physical properties of Single Event Upsets (SEUs). Then we present a model of a simple processor with the applied scheme as an example. The scheme is extended to support n-way simultaneous multithreaded systems, resulting in two basic schemes. A cost analysis of the proposed SEU-detection schemes leads to the conclusion that both schemes are applicable at reasonable costs for pipelines with 5 to 10 stages and maximal 4 hardware threads. A worst-case simulation using software fault-injection of transient faults in the processor model showed that errors can be detected with an average of 83% even at a fault rate of 10^{-2}. Furthermore, the scheme is able to detect an error within an average of only 5.05 cycles.

1. Introduction and motivation

The growing number of applications for computing systems led to rapidly growing performance requirements. Performance is easily gained by accelerating the clock frequency F. It is inevitable to decrease the main current at high clock rates, because the power consumption $E \sim F^3$ [16]. To decrease the energy consumption, the current potential between logic ‚0' and ‚1' is reduced. By increasing the integration density, new algorithms and paradigms can be implemented in hardware and energy consumption can be reduced again. Examples are dual-core systems or Simultaneous Multithreading (SMT) [1]. The Semiconductor Industry Association (SIA) roadmap shows an increase of integration density to 22 nm until 2016 (http://public.itrs.net/Files/ITRS_Overview.pdf). At 90 nm and below [11] a problem occurs at sea-level, which was only known from aerospace applications: The collision with high-energetic neutrons from deep-space with silicon. At larger heights, the fault rates in SRAMs increase by a factor of 3-10, approximately by 1.3 per 1000 ft [12].
These so-called total ionizing dose effects are caused by the influence of electromagnetic waves or particle radiation, being able to ionize atoms or molecules so that electrons are removed. The loose ions are very reactive and can cause severe circuit damage. Electromagnetic radiation can cause ionization if the wave-length is below 100 nm, since the photon has enough energy to separate one electron. Single Event Effects (SEEs) [6][7] are caused by interaction of a single particle with silicon. They have been investigated since the late seventies, leading to the discovery of memory faults in terrestrial [5] and extra-terrestrial [4] environments. SEEs can be separated in non-destructive *soft-errors* (or transient faults), causing a temporal malfunction or disturbance of digital information and destructive effects causing permanent failures. With high integration, protons are

1-4244-0054-6/06/$20.00

able to induce Single Event Upsets (SEUs), leading to a higher SEU susceptibility of the concerned circuits, especially for deep-space applications. R. Baumann showed that the downtime costs caused by SEUs have increased dramatically [27]. It is forecast that the *soft-error-rate (SER)* in combinatorial circuits will increase approximately by 10^5 from 1992 until 2011 [13]. Thus, we have to deal with a total soft error rate of 10^4 FIT (Failure in Time=1 error in 10^9 hours of operation) $\Rightarrow$ fault rate $10^{-5}/h$ in combinatorial circuits for the next decade [13]. Therefore it is necessary to secure the execution of future processors, making them more reliable to face the increasing number of SEUs.

This paper makes the following contributions:

- It introduces an error detection scheme which computes checksums out of the control path to detect transient errors and proposes extensions to support SMT.
- We will estimate the area for all proposed schemes and present the results of a fault-coverage analysis based on software-implemented fault-injection.

The rest of the paper is organized as follows: We will have a look at related work in Section 2. Section 3 discusses the fault model and important SEU-properties. Section 4 presents two schemes to secure the pipelined execution and Section 5 an estimation of the costs. Section 6 discusses the simulation methology and presents the results of the fault-coverage analysis. Section 7 concludes the paper.

2. Related work

Pipeline signatures resemble control-flow monitoring techniques, where incorrect branches are detected. The software-implemented execution-checking of loop-free intervals by [18] could detect all sequence errors that resulted in a branch outside the interval. Macroinstruction control-flow monitoring divides the application program into several blocks. The blocks are checked instruction by instruction for control-flow faults [19][20]. In [21] signature instruction streams (SIS) were introduced. The CRC of the instruction stream is inserted into the binary code after a branch. The monitor reads the CRC and compares it with the computed CRC. An error is detected if the checksums do not match. With a probability of a branch occurring every forth to tenth instruction, the overhead to store the signatures was between 10% and 25% of the original program code. Because the monitor is much simpler than the processor it monitors, performance degrades (because of extra memory cycles). The effectiveness of SIS was verified by hardware fault-injection for a Motorola 68000 system [21][23]. SIS raised the error detection rate to 25% in comparison to the original system. Smolens et al. [17] proposed an error detection scheme called *fingerprinting*. Fingerprinting summarizes the execution history of a processor. By using two processors, errors can be detected by comparing the fingerprints. In contrary to all schemes from above, we are able to detect errors in the pipelined execution and save hardware by using SMT. As a consequence, we do not have to replicate hardware to achieve structural redundancy or insert checksums in the instruction stream. Since the detection works very fast, counter measurements like error recovery will work more efficiently (time/ cost/ power consumption).

3. Physical effects and the fault model

Temporal errors are the main cause for errors in semiconductors. They are difficult to locate in time, because they are not always in the state causing the error. The possibility for a temporal error is 5 to 100 times higher than for a permanent error [14]. Temporal errors must not be repaired, since the hardware is not physically damaged. Apart from radiation, they can be caused e.g. from power fluctuations, loosely coupled units, timing-faults, meta-stable states and environmental influences (temperature, humidity, force).
Seldom discussed in computer science literature are important properties of SEUs like creation, duration and energy level. These properties help to understand the effects caused by particles to find appropriate counter measurements. Defect-types in silicon lattice are - amongst others - characterized through different, discrete energy levels in the band gap, the entropy-change factors, and annealing temperatures at which the bonds break. Along its path through silicon latter, an ionizing particle creates electron-hole pairs through Coulomb-scattering. The energy of the incident particle can be measured through the energy loss on its path by dE/dx in units of keV/µm or linear energy transfer (LET = dE/ρdx) in units of eV cm^2mg^{-1} (ρ is the density of the matter) or in charge per unit length (pC/µm). For silicon, 3.6 eV are needed to create an electron-hole pair. The charge-collecting dynamic has a fast (O(100ps)) and a slow (O(ns)) component [8][9]. It is important to know the duration of a transient fault, when fault-detection/ correction schemes are working on a tight time-basis as in this work. The effects of a SEU could last longer than it takes to correct the error, misleading the correction in the direction that a permanent fault occurred or that no error is detected. To determine the duration of a particle-induced transient fault, we consider citing [23]:

> „*The prompt charge is collected in much less than 1 ns, which is shorter than the response time of most MOS transistor.*“

In [24] it is shown by using a 0.6µm-CMOS process that the slow component of the charge-collecting dynamic of a SEU is active over 0.5 ns only in the substrate (at maximal 0.5 mA). The quick component of this dynamic is active for ≤0.2 ns (over 3 mA). Targeting an FPGA implementation,

this will be of no concern for modern FPGAs, since their clock cycle is still well above this limit. For further details on fault types and rates in submicron technologies, see [10]. 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. In [15] it has been shown that this modeling matches closely the real faulty behavior.

4. Pipeline signatures

In this Section we present a scheme to compute signatures on a micro-architectural level for the control path of a simple microprocessor, exploiting the pipelined execution scheme. The scheme will be extended to support SMT. Let P be a p-stage pipelined, (superscalar) processor with a finite instruction set of B≠∅ instructions and $t \in \mathbb{N}$ sets of instruction streams

$$I_1 = \{i_{1,1},...,i_{m,1}\} = ... = I_t = \{i_{1,t},...,i_{m,t}\} \subseteq \wp(B),$$

where $\wp(B)$ is the power set of B. We assume two redundant RAMs with equal code and data contents. Thus, we set t=2 (although it is possible to have multiple threads reading from one RAM) and it is $I_1=I_2$ in the fault-free case. Instruction streams do not have to be necessarily finite, because of program loops. Each stage includes a storage where the processor saves the thread-ID and the control part of the instruction being processed in that stage. For the fetch stage, the control part will be the fetched instruction. For the decode stage, it will be the part of a microprogram, driving the execution unit(s), etc. The signature computation involves the well-known cyclic redundancy check codes (CRCs) [2][3]. Cyclic binary codes are a subgroup of linear codes. They are codes with a fixed number of words 2^m and a fixed word length n where m≤n over the alphabet $x_i \in \{0,1\}$. 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 selection of a generator polynomial g(x) is not a trivial task. It must be chosen in a way that enough code words are produced and the Hamming [22] distance

$$a,b \in \{0,1\}^n ; d_H(a,b) := \sum_{i=1}^{n} a_i \leftrightarrow b_i$$

is maximal. For example, if $g(x) = x+1$, all single-bit errors can be detected since g(x) is equivalent to the computation of the parity of v(x). The situation is different with pipeline checksums. Here, the message v(x) is composed out of the instruction stream and the contents of pipeline latches containing the control information for a stage. To clarify this, we start with the computation of a signature for a simple pipeline (Figure 1).

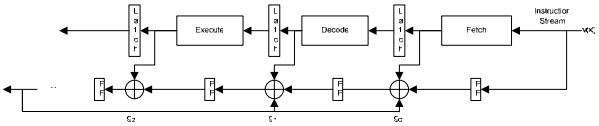

Figure 1. Signature computation

The checksums will be saved in a special memory with the corresponding PC and thread-ID. We call this storage the *checksum memory*. Table 1 shows one checksum memory entry with the according bit ranges.

Table 1. A checksum memory entry

Thread-ID	Program counter	Checksum
37	36:5	4:0

We can already apply multithreading at a coarse-grained level for this pipeline. We switch the processor context if latency-causing instructions (e.g. branches) are encountered. Branch instructions within instruction streams will lead to the storage of the checksum and to a selection of another instruction stream. If the second checksum entry is created with the same PC, the checksums are compared. If the entry is not found, it is assumed that a fault corrupted one of the PCs and an error is signaled. If the checksums are equal, no fault occurred or the checksums were changed by a transient fault in a way that both checksums are now equal. If the checksums are not equal an error will be signaled. From the calculation of checksum parts concerning a single stage from Figure 1, we see that different generator polynomials g(x), h(x) can be applied to compute single checksum bits (Figure 2). For a cost-effective implementation we considered only one generator polynomial for all pipeline stages.

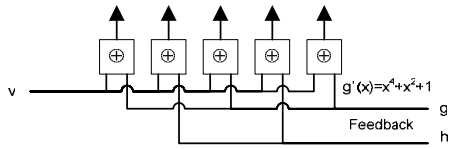

Figure 2. Checksum calculation in a stage

In fact, we have a two-level polynomial scheme applied if we regard different feedback stages. The implementation of

error correcting codes checking each latch in a pipeline stage is possible, but was omitted due to the high additional power consumption and performance loss. Parity computation for each pipeline latch will also affect performance, since we have to build the parity for all signals from the latches of a pipeline stage. This number can be quite large (e.g. signals from the microcode) so we have to build fan-in trees to compensate fan-in effects, leading to a slowdown. We also 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 scheduling will lead to different parts of the control and data stream exiting the execution stage at different times. If the fetch and execution policy is done on a cycle-by-cycle basis, we can realize this part easily, if we choose the generator polynomial in a way that no feedback affects the out-of-order stages. Since XOR is an associative operation, dynamic execution will not affect the checksum. For flexibility we want to use any fetch and execution 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. The solution is based on a two-level scheme. Two checksums are calculated separately for the out-of-order and other stages. Figure 3 shows the checksum calculation including the out-of-order stage.

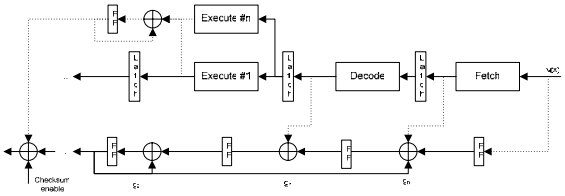

Figure 3. Out-of-order checksum calculation

For clarity, the control paths are marked with dotted lines. Both checksums are stored in FIFO buffers. Results from the execution stages are XORed until *checksum enable* is set. Then both checksums will be XORed. So we shifted the time dependence to the last stage. The scheme will not increase the costs except for the XORs in the last stage. Since this applies to all following schemes, we will not mention the costs for the final stage explicitly.

5. Cost analysis

The number of XOR gates per stage is equal to s_{max}. Typically it is $s_{\max} = \max\{s_1,...,s_p\}$, where s_i is the number of (control) wires from stage i to stage i+1. Let p be the number of pipeline-stages, t the number of threads, b the width of instructions and d the depth of the FIFO buffer from stage to stage for each instruction stream. To get an upper estimation for the costs, we assume $\forall i \in \mathbb{N}.g_i = 1.g(x) = \sum_{i=1}^{p} x^i$ for the generator polynomial. This means that the last stage is connected to all previous stages. For simplicity, we set the instruction width to b=2*32 bit, the average control path width of pipeline stages to s_i=64 bit and the FIFO-depth to d=4. The gate costs for the scheme in Figure 1 are relatively low. Using Table 2 they compute to ($\forall i.s_i = 64$):

$$C(PIPECRC_1(p,d)) = \sum_{i=1}^{p} s_i C(XOR) + \sum_{i=1}^{p+1} s_i C(FF)d$$
$$= 4\sum_{i=1}^{p} s_i + 32\sum_{i=1}^{p+1} s_i = 2304p + 2048.$$

Since the n-to-m switch will be used in the following estimations, we explicitly calculate the costs. A n-to-m switch will direct the input x[n-1:0] (width n) to one of m outputs y[n-1:0]. All other m-1 outputs will be set to zero. The output is selected by s[ld(m)-1:0]. For the number of NOT-gates, we need the number of zeroes within a binary number of length s. This can be easily calculated recursively, if we consider the following: Let s be the number of digits of a binary number. Then 2^s binary numbers are possible, $2^s/2$ beginning with zero. The remaining zeroes are two times the number of zeroes of the binary number with s-1 digits. This is:

$$_{\#}Zero(0) = 0;\ _{\#}Zero(1) = 1;\ _{\#}Zero(2) = 4;$$
$$_{\#}Zero(s) = \frac{2^s}{2} + 2\ _{\#}Zero(s-1).$$

The solution of this recurrence is:

$$_{\#}Zero(s) := 2^s\ _{\#}Zero(0) + 2^{s-1}s = 2^{s-1}s\ .$$

Thus, the cost for an n-to-m switch is (m is a power of two):

$$C(dec(n,m)) = n\left(m2^{m-1}C(NOT) + 2^m C(AND)\right)$$
$$= n\left(m2^{m-1} + 2^{m+1}\right).$$

Analogously the cost for an m-to-n switch, selecting one signal group x[n-1:0] of width n out of m groups y[n-1:0] is:

$$C(mplex(n,m)) = C(dec(m,n) + nC(OR)$$
$$= n\left(m2^{m-1} + 2^{m+1} + 2\right).$$

Figure 4 shows the signature calculation for a two-way SMT-system. It can be seen, that hardware costs double (at least) for each hardware thread. Activation and propagation signals for the checksums are not shown for clarity. The checksum will be calculated depending on which thread is

active. Each part of the checksum is activated by the thread-ID, indicating which thread is active in a stage.
Since the processor is working on the same data and code, the checksums will not be different in the fault-free case. The additional gate cost for a t-way multithreaded pipeline execution scheme in reference to Table 2 calculates to

$$C(PIPECRC_2(p,t)) = t \cdot C(PIPECRC_1(p,4))$$
$$+\sum_{i=1}^{p} C(dec(64,t)) + C(mplex(64,t))$$
$$= t\left(2304 \cdot \mathrm{p}+2048\right)+2^{\mathrm{t}}\left(128 \cdot \mathrm{p}+32 \cdot \mathrm{pt}+128+32 \cdot \mathrm{t}\right)+128.$$

Table 2. Gate cost and delay (from [26])

Gate	Cost	Delay
NOT	1	1
NAND/NOR	2	1
AND/ OR	2	1
XOR/ XNOR	4	2
Flip-Flop (FF)	8	4

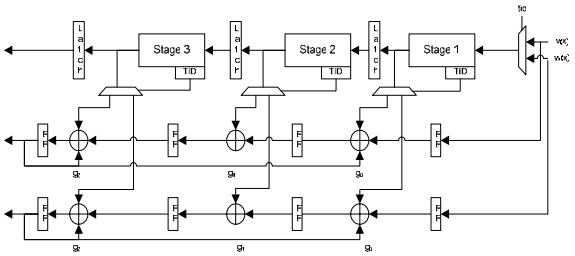

Figure 4. Checksum calculation for two threads

To compare the calculated checksums in a multithreaded system, t context switches have to occur (t is equal to the number of hardware threads). If the execution was fault-free, the same number of instructions has been executed. Then all FIFOs will have the same contents. Transient faults in the checksum mechanism will lead to different checksums and to a detection of the error. If instructions are pre-decoded, a branch - the criteria for a context change - can easily be recognized. At this point, instructions of other threads may be in the pipeline. We will have to wait for these instructions to exit the pipeline to compute the checksum. To do this, we use a change in the thread IDs in the last stage to initiate a checksum comparison (*checksum enable*). Additionally to the scheme presented in Figure 4, the scheme in Figure 5 tries to save XOR-gates, since this number can be quite large.
The thread IDs in Figure 4 and Figure 5 will assign a part of an instruction in a stage to a signature. Therefore faulty thread IDs will be detected, because the wrong signature will be selected. Then instruction streams will have different contents and lengths.

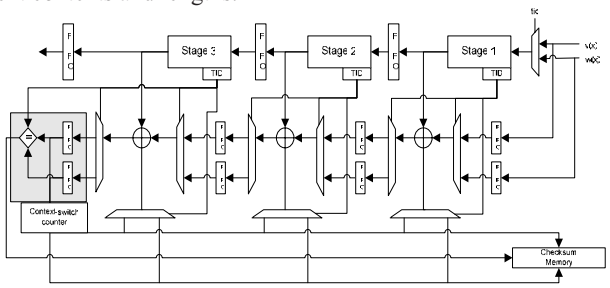

Figure 5. Extended checksum calculation

The costs for this kind of checksum calculation compute to:

$$C(PIPECRC_3(p,t)) = C(mplex(64,t)) + t\sum_{i=1}^{p+1} 64C(FF)4$$
$$+\sum_{i=1}^{p}\left(C(dec(64,t)) + 2 \cdot C(mplex(64,t)) + 64C(XOR)\right)$$
$$= 128 \cdot 2^t + 64 \cdot 2^{t-1}t + 128 + 2048t \cdot (p+1)$$
$$+ p(384 \cdot 2^t + 192 \cdot 2^{t-1} \cdot t + 272).$$

The contour plot in Figure 6 shows the difference Δ of the cost functions $C(PIPECRC_3(p,t))$ and $C(PIPECRC_2(p,t))$ for the checksum schemes in Figure 4 and Figure 5. The x-axis shows the number of hardware threads t, the y-axis the number of pipeline stages p and the z-axis the costs. We see that the costs for the scheme from Figure 4 are always lower than those from Figure **5**. Both schemes are applicable at reasonable costs for pipelines with 5 to 10 stages and a maximal number of 4 threads.

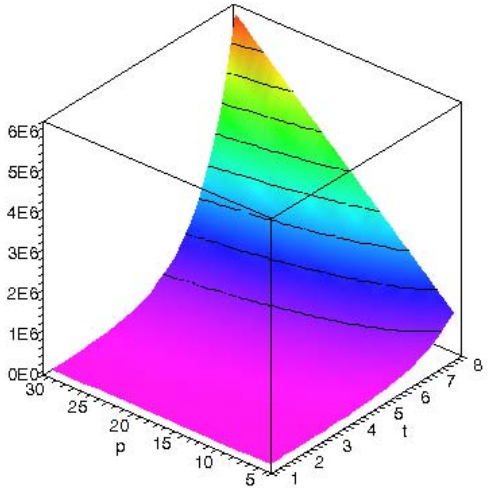

Figure 6. Contour plot of cost function Δ

6. Fault-coverage analysis

For the software simulation, we generated a random stream of 1000 32 bit instructions, which was used as an input for the modeled processor. In the first experiment we wanted to determine the best polynomial to detect an error. Branches were created with probability p_{branch}, assessing the number of instructions between checksum comparisons. The probabilities for a branch in Table 3 were gained from SPEC95 benchmark simulations by using SimpleScalar [25].

Table 3. 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 computed the checksum for a 32 bit instruction stream without fault. Then we simulated transient faults in the second instruction stream by flipping single, randomly chosen bits at random stages with a fault rate of 10^{-2}. We chose such a high error rate to speed up fault-injection experiments. This was done for 1000 fault injection runs. In each fault-injection run transient errors were injected. As model we selected a multithreaded 5-stage pipeline with an internal control-path width of 32 bit from stage to stage. For a worst case study, we assumed that the pipeline will be flushed each time a fault is detected or a branch is encountered. On a branch in the second instruction stream both checksums were compared. Due to its simple design, we chose to simulate the checksum scheme from Figure 4. Figure 7 shows the results for the fault coverage analysis to find the polynomial with the best fault coverage.

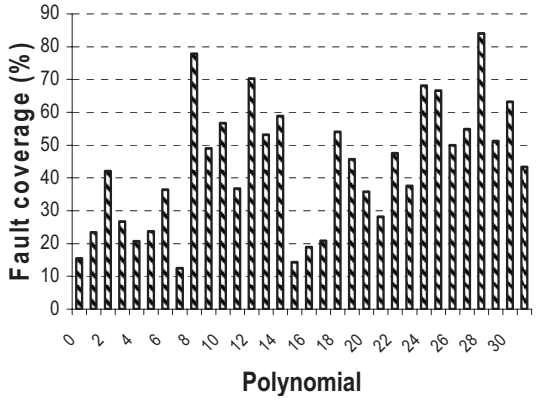

Figure 7. Fault coverage in %

Polynomials are given as numbers in the x-axis, where e.g. '28' represents the polynomial $g(x) = x^4 + x^3 + x^2$. The y-axis shows the fault coverage in %. We conclude from Figure 7 that the best fault coverage is achieved by applying the polynomial $g(x) = x^4 + x^3$ (83%). Figure 8 shows the fault coverage in relation to the probability of a branch in % for g(x). We see that the fault coverage is strongly dependent from the number of branches. The probability for a branch was chosen to range from 0.2 to 0.0032.

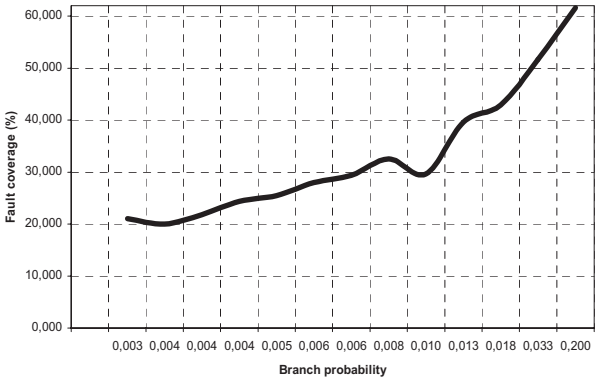

Figure 8. Fault coverage-branch relation

But how fast are errors detected? To find an answer, the gained polynomial was used to compute the checksums in the second step of the analysis. p_{branch} was set to the upper average of the values from Table 3 (20%). As the number of branches substantially determines the number of checks, errors will be detected after $\frac{2}{p_{branch}} \geq 2n$ executed instructions (two instruction streams generating checksums). Figure 9 shows the experimental results - the latency in cycles to detect an error. Note that 'Time' on the x-axis is a non-linear factor, since errors occur randomly. The high latency at the beginning results from the initialization phase of the scheme. Since the pipeline is cleared on every branch, this affects the fault coverage and latency, since a feedback with zero does not result in a checksum with high fault coverage.

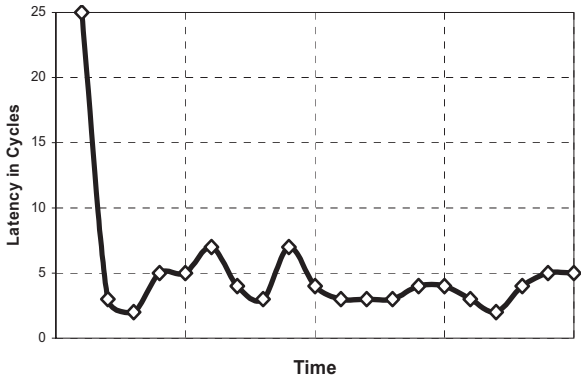

Figure 9. Latency in cycles to detect an error

The average number of cycles to detect an error was computed to 5.05.

7. Summary and Conclusion

In this paper we presented a scheme to detect transient errors in pipeline stages of a microprocessor by fetching from two RAMs with identical code and data contents and calculating a checksum using a generator polynomial. Checksums are compared on every second branch. Since branches occur with an average probability of approximately 20% in the instruction stream, checksums are compared often enough. The worst-case analysis by using generated 32 bit instruction streams for a multithreaded 5-stage pipelined processor with an internal control-path width of 32 bit showed that an average of 83 of all injected faults can be detected – even at a fault rate of 10^{-2}. We chose such a high fault rate to speed up fault injection experiments. Overall the presented scheme is simple and efficient enough to be integrated in most contemporary microprocessors. It can detect an error very fast - within an average of 5 cycles. The redundant RAMs can be omitted if the memory is secured against transient faults by using Error Correcting Codes and the fetch bandwidth is large enough. Future work will comprise a Field Programmable Gate Array implementation and an analysis of the power consumption, size and performance.

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] G. Kane, J. Heinrich, MIPS RISC Architecture, Prentice Hall, Englewood Cliffs, 1992.

[5] T.C. May, M. H. Wodds, Alpha-Particle-Induced Soft Errors in Dynamic Memories, In Proc. of the 1978 IEEE International Reliability Physics Symposium (1978).

[6] T. Weatherford, IEEE Nuclear and Space Radiation Effects Conference (NSREC) 2002, Short Course, From Carriers to Contacts, A Review of SEE Charge Collection Processes in Devices, 2002.

[7] S. E. Kerns with contributions from B. D. Shafer, Transient-Ionization and Single-Event Phenomena, In: P.V. Dressendorfer, T. P. Ma (Editors), Ionizing Radiation Effects in MOS Devices and Circuits, Wiley, 1989.

[8] F. Faccio et al., SEU effects in registers and in a Dual-Ported Static RAM designed in a 0.25 μm CMOS technology for applications in the LHC, CERN/LHCC/99-33 (1999) 571.

[9] E. L. Peterson, IEEE Nuclear and Space Radiation Effects Conference (NSREC), Short Course, Single-Event Analysis and Prediction, 1997.

[10] T. Juhnke: Die Soft-Error-Rate von Submikrometer-CMOS-Logikschaltungen Fakultät Elektrotechnik und Informatik, Technischen Universität Berlin, Dissertation, 2003.

[11] E. Normand, "Single Event Upset at Ground Level," IEEE Transactions on Nuclear Science, Vol. 43, No. 6, December 1996.

[12] H. Kobayashi, et. al., "Soft Errors in SRAM Devices Induced by High Energy Neutrons, Thermal Neutrons and Alpha Particles," IEDM Tech. Digest, Dec. 2002, pp. 337-340.

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

[14] S.R. McConnel, D.P. Siewiorek, M.M. Tsao: The Measurement and Analysis of Transient Errors in Digital Systems, Digest of Papers, FTCS-9, pp.67-70, 1979.

[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] U. Brinkschulte, T. Ungerer, *Mikrocontroller und Mikroprozessoren*, Springer-Verlag, 2002.

[17] J.C. Smolens, B.T. Gold, J. Kim, B. Falsafi, J.C. Hoe, A. Nowatzyk: "Fingerprinting: bounding soft-error de-

tection latency and bandwidth". ASPLOS 2004: 224-234.

[18]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.

[19]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.

[20]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.

[21]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.

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

[23]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

[24]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.

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

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

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