On the Usefulness of Detecting Soft Errors in Parallel Pipelines for High-Speed Machine Vision Based on Organic Computing

Marcus Komann, Frank Taubert, and Dietmar Fey
Friedrich-Schiller-University Jena, Germany
marcus.komann@cs.uni-jena.de, fey@uni-jena.de

Abstract

Computer hardware is always in danger of being hit by ionized particles which cause different kinds of faults. In this paper, the effect of transient soft-errors on the stages of parallel pipelines is investigated by the example of emergent algorithms and machine vision. A fault injection model is introduced and the corresponding protection scheme is presented. During simulation, the number of relevant faults showed to be relatively small and the effect on the pipelines was mostly negligible. However, the costs for protection are low possibly making use of protection worthwhile.

1 Introduction

The term *Single Event Effect* (SEE) describes perturbations in semiconductor elements which are caused by ionizing radiation like, e.g. cosmic rays or neutron radiation. Such particles hit or cross computer hardware and either temporarily disturb the functionality or destroy parts of the systems. Errors which account for permanent failure are called *hard errors* while errors which only result in defects over a period of time are called *soft errors*.

Some years ago, soft errors were hardly in the focus of the reliability community [5]. But they have received growing interest recently because they are a major threat to today's electronic systems. For example, Baumann [1] compared different types of soft errors and showed that their rate in a system can be higher than that of all other errors combined. Furthermore, paying attention to these effects on computer systems is especially important in aerospace technology because the probability of being hit by a particle is much higher in upper layers of the atmosphere due to higher radiation. But awareness and possible protection is also important on sea level because SEUs also become more likely there due to ever shrinking structure sizes. Anyhow, reliability and fault-correction are a major concern of computer industry today.

Soft errors divide into *Single Event Upset* (SEU) and *Single Event Transient* (SET) errors. SEUs produce bit-flips in memory. SETs result in undesired changes of signal levels. The reason for such transient errors are particles crossing the semiconductor which emit energy in a so-called *Line Energy Transfer* which produces a change in the electric charge of the component. These errors thus don't destroy the component and can be reverted if they are detected. See [1] and [5] for an overview and more details on the subject.

Such failures in technical systems might not have any impact on the processing of some problems because they alter irrelevant memory/signals or because their wrong value is overwritten and thus corrected in later steps. But in some scenarios, soft errors might have severe consequences ranging from simple production facility breakdown to danger for human lives as in airplanes or cars. The main question in general is thus: Which effort is worth the protection of which system? In this paper, we want to answer this question towards an agent-based, massively-parallel, high-speed machine vision system and transient errors. Protection costs time and money but might pay back the effort with robustness, effectivity, and longevity of the system.

The paper is structured as follows. We present the Organic Computing vision system and the corresponding pipeline architecture in Sect. 2. In Sect. 3, the fault model we used is introduced and the scheme of fault injection into the architecture is shown. At last, we wrap up, discuss the results, and take a look forward in Sections 4 and 5.

2 The Marching Pixels concept and the corresponding pipeline architecture

This section presents the architecture in which the faults are injected in Sect. 3. At first, the Marching Pixels project is motivated and described briefly. Afterwards, the pipelined version of the corresponding architecture is explained.

2.1 Marching Pixels - Massively-parallel machine vision

In modern factories, machine vision is an elementary part of the production processes. Robots have to see in order to grab, drill, grind, or in general handle tools and work pieces. With increasing production speeds, the need for extremely fast vision systems is ever-growing. On the other hand, it is also important that these vision systems don't become too large for reasons of power consumption, price, or simply limited space for installation, e.g. on a robot's gripper arm.

One of these vision problems is detection of objects and their attributes in binary images. In this case, detection means classification of objects, which for example lie on an assembly line, according to a set of pre-known object categories. It also means detection of defective or in-

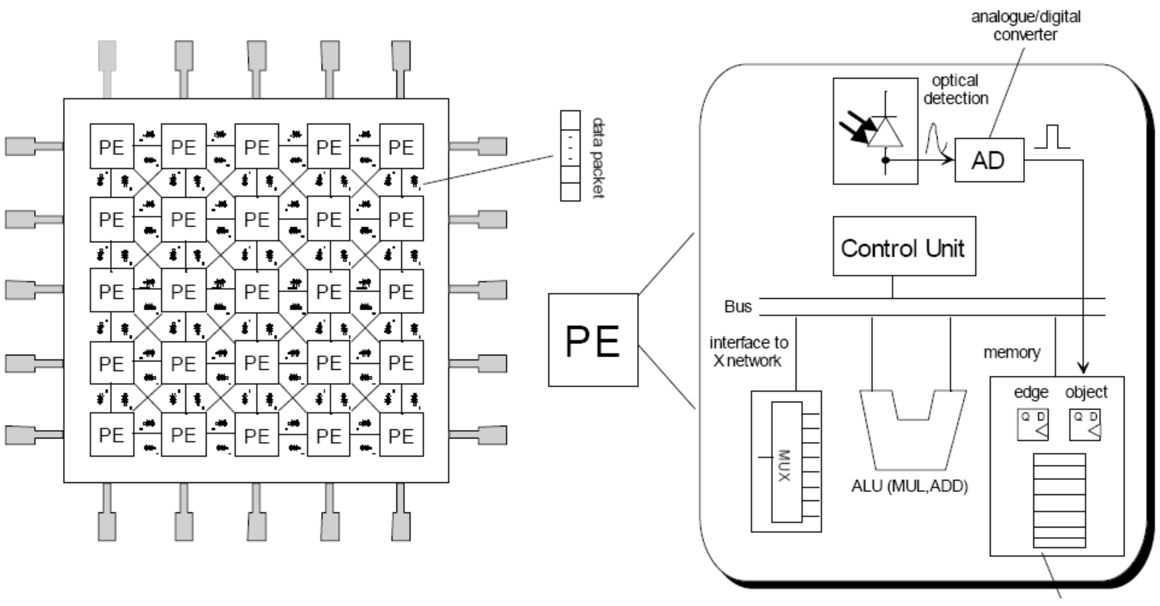

Figure 1: Architecture of processor element array and of a single PE

complete work pieces which should be sorted out by the production system. Apart from object classification, it is also important to detect some properties of the single objects like size, edge lengths, rotation, and centroids in this use case. All of these tasks shall furthermore be carried out for multiple objects per image and possibly thousands of images per second in order to keep the productivity of the assembly line and its corresponding factory high.

When such opposing aims of low size and high speed meet, classic serial vision architectures reach their limits. This is especially true if the systems don't have to detect one but many objects at a time whose number is not known in advance. A practical example for this are small pieces like nails or screw-nuts lying on the mentioned fast moving assembly line.

One system architecture that is able to cope with these opposing aims is developed in the *Marching Pixels* project. It is explained shortly in the next paragraphs. For more details on the system, the algorithms, capabilities and weaknesses, and implementations in VHDL and FPGA, please refer to earlier publications like [3], [4], [6], or [7].

Put very short, the basic idea of Marching Pixels is to take a binary image, load it into a massively-parallel field of simple processing elements, and to let agents, so-called Marching Pixels (MPs), travel around this field in order to detect objects and their attributes. These agents work collaboratively and thereby exploit emergence [10] to fulfill their goals. The advantage of this approach is that the complete architecture can be implemented on a relatively small dedicated chip and thus in a small embedded vision system (including camera and input/output). The massive parallelism of processing elements grants large compute power which can be used to meet strict realtime requirements.

This description proposes dividing an MP vision system into two parts. The first one is an architectural layer where the field of processing elements in a whole, the processing elements themselves in detail, and the input/output characteristics have to be defined. The second part is an upper layer where specific emergent algorithms that are executed on the single processing elements have to be found and tested. This is the level where the Marching Pixels agents are running.

2.1.1 Architectural layer

In this layer, the hardware structure necessary for emergent agents is described. Figure 1 shows the principle. On the left side, the processor element (PE) array on a chip can be seen. The PEs have a specific local neighborhood. In the figure, the PEs have Moore neighborhood with radius=1. But also von-Neumann neighborhood and neighborhoods with larger radii are possible. PEs don't have connectivity across the borders. Outer PEs are used for outside communication instead. But also

On the right side of the figure, you can see a PE in detail. It has an optical detector on top which captures a part of the image, i.e. one pixel in the simplemost case. This pixel is then binarized and used as input to digital processing. In the digital part, arbitrary digital operations can be used like binary logic, shift operations, multiplexing, or even arithmetic operations in ALUs. Apart from that, it is possible to use flip-flops as local memory for each single PE.

The agents can now be described as data packages of states and memory which are moved between different PEs. The state machine/program of the agents is modeled in digital hardware and the states of the agents are stored in local memory. This system can also be used for fast parallel execution of other local operators as used in cellular automata, local image filters, and so on.

2.1.2 Algorithmic layer

Having PEs with these abilities, we now can think of algorithms which solve the previously described problem of fast object detection in binary images. In the Marching Pi-

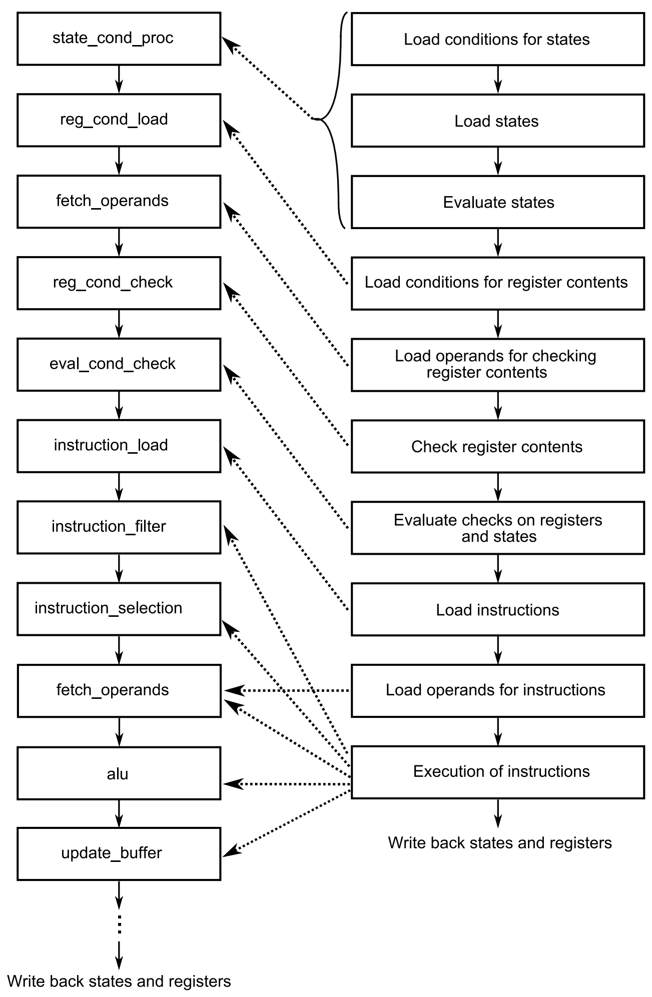
state_cond_proc
reg_cond_load
fetch_operands
reg_cond_check
eval_cond_check
instruction_load
instruction_filter
instruction_selection
fetch_operands
alu
update_buffer
Write back states and registers
Load conditions for states
Load states
Evaluate states
Load conditions for register contents
Load operands for checking register contents
Check register contents
Evaluate checks on registers and states
Load instructions
Load operands for instructions
Execution of instructions
Write back states and registers

Figure 2: Pipeline stages and their tasks.

xels project, we orient on ant-like behavior to steer agents across the PE array (and thus the image) with the goal of visiting certain pixels and thereby compressing gathered information. In the end, the centroid pixel shall be found and desired information like size, rotation, or edge lengths shall be output.

On their way, these agents called Marching Pixels (MPs) can mutually interact directly or via indirect, so-called stigmergic, communication. MPs can be born, can unite, and they can die. Exploiting these capabilities, we are able to create several emergent object detection algorithms with different capabilities and requirements. We won't go into detail about the specific algorithms here because this paper is on the pipelined implementation of the hardware. Please refer to the mentioned literature for details.

2.2 From an array of processor elements to pipeline execution

Implementing possibly several thousands of PEs on a chip or FPGA is expensive and sometimes impossible due to size constraints. It might therefore be useful to trade some of the execution time for architecture size if given (real)time requirements can still be met afterwards. Using time-multiplexing pipelines which execute several PE positions/cells[1] one after another makes it possible to execute virtual agent schemes with a smaller hardware requirement in longer time. For example, for a *n*n* image, using *n* parallel pipelines, one for each line, takes *O(n)* times longer for execution but saves *O(n)* space keeping in mind that pipeline execution costs some overhead.

In [9], a generic pipeline architecture for emergent algorithms in general and Marching Pixels algorithms in detail was specified in VHDL. Generic means that most of the pipeline properties can be chosen freely, such as number of parallel pipelines, size of the array (for MPs: image), number and size of registers for states and variables, neighborhood, or data width. Apart from that, the state transition of each position/cell in the array is passed to the system via a rule table. And at last, a set of instructions along with specific conditions can be defined which is executed at each position/cell in each step. Using state transition table and commands, it is possible to implement arbitrary parallel automata/programs. As mentioned, it could for example be used for fast simulation of Conway's *Game of Life* or for cellular automata in general [11].

Figure 2 shows the architecture of one pipeline in detail. On the left side, the pipeline stages can be seen. The tasks of the single stages are shown on the right side. For all parts which check or evaluate states or registers, not only own memory is read but also memory of the defined neighbors. Conditions and transitions can thus be dependent on information of the current position/cell and its neighbors. The same holds for the execution of instructions or arithmetic operations which might or might not be done in every pipeline cycle. Write back from the buffers is done in the end because agents and empty positions/cells need data from the previous synchronous execution step like, e.g. in cellular automata.

For example, if the current cell does not host an agent, it doesn't calculate anything but simply waits in a defined state until an active agent is crossing its neighbors. When an agent moves towards the cell, the cell takes the agent by changing its state to the agent's state and copies the agent's memory in its own memory. The original cell of the agent deletes all information and turns to a passive state. The new cell of the agent might then furthermore calculate some values using the ALU if certain conditions are met, e.g. if the cell is the final cell or an edge cell.

3 Redundant pipeline buffers and fault injection

3.1 Majority voting of duplicated buffers

Giving a short reminder, the question we are trying to answer is if it is worth protecting pipeline stages as described in Sect. 2.2 against errors? We do not want to identify the specific circuit in which a fault occurs. Instead, we are try-

[1] One could also call the PE positions cells of a regular array as in cellular automata.

ing to figure out which pipeline stage as a whole unit is vulnerable and thus insert the detection and protection modules between the pipeline stages.

In this paper, we focus on answering this question concerning transient errors (SETs) as shown in Sect. 1. Bit flip errors (SEUs) are less interesting and not scope of this paper because they occur in memory and can be handled relatively easy with normal error correction codes. Transient errors alter signal levels in logic temporarily and thus can create wrong results of the whole system if they are not detected and corrected.

We now look at the described pipeline and try to find out which stages of it are vulnerable and how severely faults occurring in the stages change the output of the whole system. With the proposed pipeline extension, errors in stages are not only detected but are furthermore instantly corrected. The write back part of the pipeline does not need protection because it writes memory directly and thus has no logic which can be disturbed. But vulnerable logic is situated in each of the pipeline stages.

Between every two stages, buffers catch the results of the prior stage and memorize them. The result values then serve as input for the following stage. Those buffers are redundantly duplicated two times each. Together with the initial buffer, we end up with three buffers, which are used to detect temporary errors in the stages. This approach is similar to triple modular redundancy (see also the works of Nicolaidis [8]) The basic idea is now to direct the signal from a stage to the redundant buffers one after another in order to detect errors by comparing the buffer contents in a majority voter. Therefore, the pipeline stages have to be changed to asynchronous logic first. Figure 3 shows the tripled buffer, the voter, and the required signal lines. The original buffer is at the right side. The original signal is compared to the signals at times *Clk*-Δ and *Clk-(2*Δ)* which are saved in the other two buffers. Δ is defined as the time of a transient error and is architecture-dependent. The voter decides for the correct values by making a majority decision between the three input signals effectively correcting all single transient errors. The result is written in the original buffer at time *Clk*. It is then available as output.

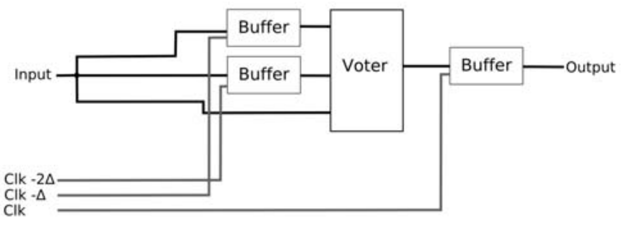

Figure 3: Multiplied buffers and corresponding clock signals.

The voter itself is not protected. While this is no problem for our test scenarios here in which we want to identify vulnerable pipeline stages, this can become an issue when the presented protection scheme is used in a real system. Faults occurring in the voter would probably lead to erroneous system behavior. However, the probability of a cosmic ray hitting the voter is very small. But for systems like airplanes, this should be taken into account. Also protecting the voter/control system is a sophisticated task and shall not be discussed here.

In this paragraph, we compare technical details of the presented protection scheme and the protection of Nicolaidis [8] because they differ in some aspects. Nicolaidis uses three latches to store intermediate values. Then, those three values are compared by a voter and the majority result is written in a fourth latch. For saving chip area, we leave out one of those latches and compare two buffers with the current signal. This change has some implications on the time behavior of the system. In our architecture, the delay between *Clk*-Δ and *Clk* not only has to be larger than a transient error. It also must be larger than the time required by the voter. The original architecture of Nicolaidis achieves voter time independence by practically delaying the result by one clock cycle. For our approach, the implementation of the voter has to be very fast as a result. Otherwise, the time of the voter would slow the computation because it would work as a lower boundary for the clock rate.

3.2 Injecting faults by altering signal levels between pipeline stages

After defining the error detection architecture, a fault injection model has to be implemented. We only look at single faults which change one bit position of a signal. [2] contains a harsh real world test where no two bit faults occurred concurrently. [13] estimates the probability of two bit faults occurring at a time as very small. In order to find weak points of the architecture, a module is inserted between the logic of each pipeline stage and its buffer. This error module is able to alter the signal level just as a particle does.

Three Marching Pixels algorithms called *Gaphop*, *Flooding*, and *Opposite Flooding* from [9] are used as test scenarios (see [6] for details on the algorithms or [12] for some sample simulations). As input, four different images with different objects are processed. The first three images contain one square object which occupies resp. 25%, 50%, and 75% of the image pixels. The last image contains several small objects.

An error simulation then worked as follows. Every image was simulated with each algorithm for at least 100 times and the errors in each stage were counted. We used an extraordinary high error probability of 50% in each clock cycle in order to find stages which are weak. After the initialization of the pipeline had finished, a fault was injected at a random stage of a random pipeline. Fault injection was then stopped until the error had traversed the pipeline because of the reasons concerning double errors mentioned above. Another reason for not analyzing two-bit errors is that we wanted to be able to assign final errors of the complete architecture to the stage and the exact fault that was injected

After injection, the errors were on one hand partitioned into relevant and irrelevant ones, and status or register errors on the other hand. Relevant errors result in wrong final image processing (creation of false or deletion of correct final states of the agents) while irrelevant ones have no effect on the final output of the system. Status errors are basically also register errors but have a larger impact because they hinder steering the Marching Pixels agents correctly while registers "just" memorize values the agents gathered during their run. Relevant register errors result in wrong register values of final state agents. This is also a drawback but the agents at least took the correct path and ended in the correct position/cell then.

4 Results

Table 1: Number of relevant errors occurring and corresponding protection costs (in UMC cell units).

	Errors			Costs	
Stages protected	State	Register	Sum	relative	absolute
none	1189	1178	2367	515285	100,00%
eval_cond_check	554	545	1099	516056	100,15%
instruction_selection	237	263	500	535731	103,97%
state_cond_proc	209	142	351	537927	104,39%
reg_cond_check	149	136	285	549712	106,68%
fetch_operands	33	78	111	574228	111,44%
Alu	0	14	14	589974	114,49%
instruction_filter	7	0	7		
instruction_load	0	0	0		
reg_cond_load	0	0	0		
all	0	0	0	606875	117,77%

Table 1 shows results of the tests and of the synthesis of the pipeline architecture. The error part shows the absolute number of relevant errors that occurred in the different stages. The cost part of the table shows how expensive it is to protect the stages from errors. It was created by synthesizing the hardware description with a *90 nm* process from UMC. This part is cumulative meaning, e.g. the costs of protecting stage `instruction_selection` include the costs for `eval_cond_check`. The numbers in the table show that the logic in the condition checking stage suffers most from errors while the ALU surprisingly doesn't. The reason for this is that the Marching Pixels algorithms don't always have to use the ALU by design. They mostly compare own and neighbors' states in order to calculate new states of the agents instead. Thus, errors in the ALU are mostly irrelevant.

Even more surprising is the small amount of relevant errors over-all. Summing up, 14407 bit-flips were injected into the architecture's inner signals. 2367 of them caused relevant errors, 824 caused irrelevant errors, and 11216 did not have any effect on the computation. The reason for this also lies in the Marching Pixels algorithms. Lots of the PE positions/cells don't have to do anything in some steps because they don't host an active MP agent. Or the cells don't ever have to do anything because they are background pixel. Background pixels are never visited by an agent in some MP algorithms. Errors in such unused cells thus have no effect on the final outcome. The same holds e.g. for errors in cells containing MP agents only at the start of a computation. If errors happen there later, they don't have any effects.

5 Discussion

In this paper, we presented how the reliability of parallel pipelines can be improved by redundancy. Therefore, buffers between pipeline stages were duplicated. Afterwards, errors were injected in the logic of the stages according to a specified scheme. A majority voting of buffer values was then able to catch and correct all applied faults. The cost for protecting the different stages was presented along with the probability of errors to cause severe system failures. This protection was applied to an Organic Computing vision system called Marching Pixels where parallel, emergent agents detect objects in binary images.

[13] estimates the number of soft errors per year to 200 on a *$2mm^2$* chip. The conclusion is that it depends whether it is worth to protect parallel pipelines with algorithms similar to Marching Pixels or cellular automata or not. It must be done if they are part of systems which should never fail, like e.g. in airplanes. On the other hand, the occurrence of errors is very improbable and the effect of errors is furthermore mostly negligible. However, the cost for protecting all pipeline stages is only 17% in space and the cost for protecting the four most error-prone stages is just over 4% making protection relatively cheap.

For the future and due to the rareness of errors, another protection architecture seems interesting where history bits are used instead of tripled buffers for comparison of signal levels. If levels differ there, the pipeline is stopped and rolled back in order to repeat the erroneous last computation step. This costs one complete clock cycle time when errors occur due to the rollback of a complete pipeline not coming for free. But it saves chip area because only the history bits and the rollback logic have to be added in hardware instead of tripled buffers and voters as presented in this paper.

The two approaches' time constraints can be compared mathematically as follows. Given the error probability E, $1/E$ defines the average number of clock cycles between two errors. Let T furthermore define the unprotected clock cycle time. Protection with buffers requires a cycle time of $T+2*\Delta$ while protection with history bits requires $T+\Delta$ time if no error occurs, i.e. one Δ less. History bit protection is faster if the time for rollback if an error occurs is smaller than saved time (in comparison to tripled buffers) if no error occurs:

$$E*(T+\Delta) < (1-E)*\Delta$$
$$(T+\Delta)/\Delta < (1-E)/E$$
$$T/\Delta < 1/E-2$$

Thus, history bit protection is faster if unprotected clock cycle time divided by maximal error duration is smaller then average number of clock cycles between errors minus two.

References

[1] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability, 5(3):305-316, Sept. 2005.

[2] Damien Chardonnereau, Raijmond Keulen, Michael Nicolaidis, Eric Dupont, KholdounTorki, Fabien Faure, and Raoul Velazco. Fault tolerant 32-bit risc processor: Implementation and radiation test results. In Single Event Effects Symposium, April 23-25 2002, Manhattan Beach, California. ww.iroctech.com, 2002.

[3] Dietmar Fey, Marcus Komann, Frank Schurz, and Andreas Loos. An organic computing architecture for visual microprocessors based on marching pixels. In ISCAS, pages 2686-2689. IEEE, 2007.

[4] Dietmar Fey and Daniel Schmidt. Marching pixels: A new organic computing paradigm for smart sensor processor arrays. In CF ’05: Proceedings of the 2nd conference on Computing frontiers, pages 1-9, New York, NY, USA, 2005. ACM.

[5] Tino Heijmen. Radiation-induced soft errors in digital circuits a literature survey. Report 2002/828, Philips Electronic National Lab., Netherlands, August 2002.

[6] Marcus Komann and Dietmar Fey. Realising emergent image preprocessing tasks in cellular-automaton-alike massively parallel hardware. International Journal of Parallel, Emergent and Distributed Systems, 22(2):79-89, 2007.

[7] Marcus Komann, Andreas Mainka, and Dietmar Fey. Comparison of evolving uniform, non-uniform cellular cutomaton, and genetic programming for centroid detection with hardware agents. In Victor E. Malyshkin, editor, PaCT, volume 4671 of Lecture Notes in Computer Science, pages 432-441. Springer, 2007.

[8] Michael Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In VTS ’99: Proceedings of the 1999 17TH IEEE VLSI Test Symposium, page 86, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Marcus Wagner. Entwurf einer generischen Prozessorarchitektur für emergentes Rechnen. Diploma thesis, Chair for Computer Architecture, Friedrich-Schiller-University Jena, 2007.

[10] Tom De Wolf and Tom Holvoet. Emergence versus self-organisation: Different concepts but promising when combined. In Sven Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos, and Radhika Nagpal, editors, Engineering Self-Organising Systems, volume 3464 of Lecture Notes in Computer Science, pages 1-15. Springer, 2004.

[11] Stephen Wolfram. A New Kind of Science. Wolfram Media Inc., Champaign, Ilinois, US, United States, 2002.

[12] www2.informatik.uni-jena.de/oc/english.html.

[13] J. F. Ziegler. Terrestrial cosmic rays. IBM Journal of Research and Development, 40(1):19-39, 1996.