

Fault-Tolerant Neural Architectures: the Use of Rotated Operands

Yuang-Ming Hsu*, Vincenzo Piuri§, and Earl E. Swartzlander, Jr.*

* Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA

§ Department of Electronics and Information, Politecnico di Milano, piazza L. da Vinci 32, I-20133 Milano, Italy

ABSTRACT

The use of neural networks in mission-critical applications requires concurrent error detection and correction at architectural level to provide high consistency and reliability of system outputs. Time redundancy allows for fault tolerance in digital realizations with low circuit complexity increase when timing constraints are not particularly strict: the use of time redundancy implemented via operand rotation is discussed in this paper.

I. INTRODUCTION

Validation of the network's outputs becomes a mandatory issue in designing neural architectures for mission-critical applications, e.g., in aerospace applications, in advanced manufacturing, and in critical control systems. Consistency and correctness need to be continuously monitored to avoid complete application failure due to errors generated by the neural component.

Specific configuration of some neural paradigms allows for a limited amount of intrinsic error correction capability [1], even if no information on correctness of the current network's output and on the location of the faulty component is provided. No assumption can be a priori defined of the error magnitude with respect to the normal magnitude of the network results, so that an error can be extremely critical for the application. In addition, when neural outputs are used by other parts of the system, their correctness must be determined as soon as they are generated in order to avoid the use of erroneous outputs for subsequent critical operations. Concurrent techniques must therefore be adopted at architectural level to provide fault tolerance in digital implementations of neural paradigms.

In this paper, we present the use of REcomputation with Rotated Operands (RERO), a time-redundancy technique based on circular rotation of the operands during the redundant computational phases. It is a generalization of Recomputation with Circularly-shifted Operands, a traditional variation of Recomputation with Shifted Operands proposed in [2] for error detection in serial parallel multipliers.

We consider multi-layered feed-forward networks, although this approach can be applied to any kind of neural paradigm. The operation of neuron i is defined by $y_i = f_i(\sigma_i - \theta_i) = f_i(\sum_k w_{ik} x_k - \theta_i)$, where x_k is the neuron's input from neuron k , w_{ik} is the interconnection weight from neuron k to neuron i , y_i is the neuron's output, θ_i is the threshold, σ_i is the weighted summation of neuron's inputs, and f_i is the non-linear activation function. No restriction is imposed either on the nature of the non-linear function or on the values of the interconnection weights. Operands are integer or

fractional two's-complement numbers; both the neuron's inputs and the weights are represented with n bits (minor changes are required for operands with different sizes) and are managed in bit-parallel way.

Direct-mapping architectures are derived by one-to-one mapping of neural operators onto digital components [3]. In each neuron, every arithmetic module generates one product term of the input weighted summation from one neuron's input and the related interconnection weight; this term is added to the partial summation produced by the previous module. The new partial summation is propagated to the subsequent module, until the last module fed the complete weighted summation into the circuit evaluating the activation function. Finally, the neuron's output is delivered to the receiving neurons.

The system throughput can be increased by pipelining, while the overall circuit complexity can be decreased by time-multiplexing. For simplicity, we consider only the direct-mapping solution, even though the other architectures can be easily implemented.

To abstract from the specific implementation technology, we consider the general problem of detecting (and possibly correcting) any transient or permanent error in the result of any arithmetic unit and interconnection path, whenever faults affect the computation of only one neuron or one interconnection. Any multiplicity of errors (i.e., multiple erroneous bits in the result) is covered.

II. RERO FOR CONCURRENT ERROR DETECTION

RERO consists of multiple repetition of the same sub-computations on different arithmetic components and comparison of the results. Partitioning of the operands and computation is performed at the bit level within each operand. Operands are shifted to the right (or left) for k bits during the computation phase and the result bits are shifted to the left (or right) for comparison. We set $k = \lceil n/r \rceil$, being r the minimum number of repetitions for the fault-tolerance strategy (namely, 2 for detection, 3 for correction).

In addition, both the n -bit operands are partitioned into two $\lceil n/2 \rceil$ -bit parts: the Least-Significant Part (LSP) and the Most-Significant Part (MSP). During the nominal phase, LSPs are added by the least-significant part of the adder; the result (denoted first LSP result) is generated and stored for output checking, while a carry bit is produced and propagated to the most-significant part of the adder which adds the MSP's and stores the result (first MSP result).

During the redundant phase, the two parts of the operands are circularly rotated and presented to the arithmetic unit. The LSPs are summed by the most-significant part of the adder with a zero carry input: the

result (denoted second LSP result) is available for checking. The carry bit of the second LSP result is propagated to the carry input of the least-significant part of the adder where the MSPs are summed to produce the second MSP result.

Thus the traditional adder is physically partitioned into two halves, which are connected so as to support the two alternative paths for carry propagation. A two-input multiplexer is connected at the least-significant part of the adder's carry input to select the carry in for each operation phase (namely, 0 or the carry-out bit generated by the most-significant part of the adder); a similar multiplexer is connected at the most-significant part of the adder's carry input (to choose between the least-significant part of the adder's carry-out bit or 0).

In multiplication, only one operand is partitioned, the other one is kept full-size. The multiplier number is divided in two parts (the LSP and the MSP of the multiplier). Time-redundant operations and checking are performed as in the adder, except for the higher number of intermediate carries.

In neural architectures, we extend this technique at a higher architectural level to deal with the whole arithmetic system as a single entity in order to to minimize the hardware overhead. The resulting structure for the direct-mapping architecture is shown in Fig. 1; it is composed of a set of arithmetic modules, circuits evaluating the activation function, and checkers. The internal structure of the arithmetic modules is similar to the one of the nominal architecture: the arithmetic units are modified as described above. No detection is performed at the output of each arithmetic unit. Presentation of neuron's inputs in nominal and rotated formats is performed by suited circuits in the delivering neuron. Interconnection weights are stored within each arithmetic module. Checking may be applied both to the weighted summation and to the neuron's output; checking only the neuron's output may lead to error masking due to the non-linearity of the activation function.

The linear part of the neural operation is an inner product. In the individual neuron i , each input x_k is partitioned in two parts (x_k^L and x_k^H), while the interconnection weight w_{ik} is kept full size since a multiplication is performed first.

During the nominal phase, the inner product is computed for the neuron's inputs x_k and the weights w_{ik} . In the least-significant part of the k -th arithmetic module, the least-significant portion x_k^L of the input is multiplied by the corresponding weight w_{ik} , to generate the least-significant portion $P_{i,k}^L$ of the partial term $P_{i,k}$ of the inner product. Intermediate carries of the multiplication are propagated from least-significant part of the k -th arithmetic module to the most-significant part of the k -th arithmetic module itself. In the most-significant part of the module, $P_{i,k}^L$ is added to the least-significant portion $\sigma_{i,k-1}^L$ of the partial summation $\sigma_{i,k-1}$ generated by the $k-1$ -st arithmetic module ($\sigma_{i,k-1}^L = \sum_{h=1,k-1} P_{i,h}^L$). This sum $\sigma_{i,k-1}^L$ is propagated to the $k+1$ -st arithmetic module and the carry is propagated to most-significant part of the module. At the same time, in the most-significant part of the k -th module, the most-significant portion x_k^M of the neuron's input is multiplied by the weight w_{ik} to generate the most-significant portion $P_{i,k}^M$ of the partial term $P_{i,k}$. In the

most-significant part of the module, $P_{i,k}^M$ is added to the most-significant portion $\sigma_{i,k-1}^M$ of the previous partial summation, producing the most-significant portion $\sigma_{i,k}^H$ of the new partial summation which is propagated to the most-significant portion of the $k+1$ -st arithmetic module. Outputs of the least-significant part and the most-significant part of the N -th module (where N is the number of inputs of neuron i) are the LSP and MSP of the weighted summation σ_i , respectively.

During the redundant phase, the two parts of each pair (x_k^L , x_k^M) are rotated, while the corresponding weight w_{ik} is not modified. In each neuron, the k -th most-significant part of the module multiplies the LSP x_k^L by w_{ik} , producing $P_{i,k}^L$; intermediate carries are propagated to the least-significant part of the module. In the most-significant part of the module, $P_{i,k}^L$ is added to $\sigma_{i,k-1}^L$ to generate $\sigma_{i,k}^L$ which is delivered to the subsequent arithmetic module, while the carry is propagated to least-significant part of the adder. In the least significant part of the module, the MSP x_k^M is multiplied by the weight w_{ik} , generating $P_{i,k}^M$ which is added to the MSP $\sigma_{i,k-1}^M$. The outputs of the least-significant part of the adder and the most-significant part of the adder in the N -th module are the MSP and LSP of the weighted summation σ_i , respectively.

The threshold θ_i is treated as a partial term: it is presented at the partial-summation input of the module M_i during the nominal phase, while its rotated parts θ_i^L and θ_i^M are delivered during the redundant phase.

Due to the linearity of the weighted summation and to the independence of each part of the arithmetic structure, any error in the arithmetic components is propagated towards the final summation σ_i . To provide error detection of the linear part, only one comparison is strictly required on the outputs of the circuits generating σ_i . An error is detected whenever the corresponding parts of the results generated during the nominal and the redundant phases are different.

A RERO checker is composed by a register and a comparator. The checker's input bits are connected to the register to hold the value generated during the nominal phase; the result of the redundant phase is available at the checker's input bits during the redundant phase. Rotation is performed to have both results correctly aligned by means of a custom interconnection of register outputs to comparator inputs.

Faults may occur also in the interconnection paths between neurons. To minimize the silicon area used by interconnection paths and to protect them, rotation of x_k^L and x_k^M is performed in the neuron k generating x_k ; this allows for only one full-size interconnection path between each pair of neurons and for minimizing the number of multiplexers in the neural network.

Weight registers are hard-core since they are unique; a possible solution is based on registers duplication: each multiplier is connected to a separate pair of weights through a multiplexer.

Protection of the activation function $f_i(\cdot)$ is strictly related to the kind of non-linearity. In the case of a linear function, protection is achieved as for the linear part. No checking is needed on the weighted summation since possible errors are directly propagated to the neuron's output by the activation function; for the same reason, no checking is required on the neuron's outputs (except on the final network's outputs).

If the circuit evaluating $f_i(\cdot)$ is very large, we can assume the activation function generator as hard-core. Checking is applied only on the weighted summations: faults in the linear parts are localized, while faults in the activation function generator allow for identification only of the faulty layer. For some implementation of $f_i(\cdot)$, specific techniques can be adopted to detect errors in this circuits; the linear and the non-linear parts are checked separately and fault location is always feasible. If the circuit evaluating $f_i(\cdot)$ has a limited complexity, two separate circuits can be introduced for the two phases. Each of these circuits can be specialized for receiving the weighted summation in the format of its own phase. A second approach (Fig. 1) considers two identical instances of the function generator: rotation of the weighted-summation bits is applied to obtain the same bit ordering. Checking the weighted summation is necessary to avoid masking, while checking the neuron's output guarantees the identification of the faulty neuron. Error detection is complete.

III. RERO FOR CONCURRENT ERROR CORRECTION

The approach to error detection can be extended to achieve concurrent error correction. Operands and arithmetic units are divided in three parts of $k = \lceil n/3 \rceil$ bits each. Nominal and redundant outputs are generated by three different allocations of the three operands parts on the three hardware assemblies; allocation for the redundant phases are obtained by two different rotations of the three parts of the operands, i.e., by rotating the operands by $\lceil n/3 \rceil$ and by $2\lceil n/3 \rceil$ bits, respectively. Majority voting of nominal and the two redundant results produces the correct output value.

In addition, both operands are partitioned into the Least-Significant Part (LSP), the Intermediate-Significant Part (ISP), and the Most-Significant Part (MSP). During the nominal phase, the least-significant portion of the hardware performs addition of the LSPs, producing the LSR and the carry for the intermediate part of the adder. The intermediate-significant portion of the hardware applies the same operations on the ISPs producing the ISR and the carry for the most-significant portion of the hardware, which performs the same operations on the MSPs to compute the MSR. The LSR, ISR, and MSR are stored for voting. During the first redundant phase, the operands are circularly rotated to the right and presented to the arithmetic unit producing second versions of LSR, ISR, and MSR, which are stored for voting. During the second redundant phase, the operands are again circularly rotated to the right and presented to the arithmetic unit producing third versions of LSR, ISR, and MSR. Majority voting of the three sets of LSR, ISR, and MSR selects the correct output. Unfortunately, error correction may be not guaranteed for some adder architectures, since errors may be not confined within only one part of the arithmetic unit.

Consider a ripple-carry adder. If the fault affects the generation of a sum bit without modifying the associated carry bit, error correction is ensured since only one of the three copies of each part of the result is affected by the error. If the fault changes the carry bit, the error may be propagated to the intermediate carry of

the faulty adder's part, according with the specific values of the operands' bits and, as a consequence, it may be propagated towards the other parts of the adder. For any location of the faulty adder's part, LSR can always be corrected since there are two other copies which are not affected by the error propagation. Conversely, ISR and MSR can be corrected but with a probability less than 1 since two error-free copies may not be available; by assuming a uniform distribution of operands' values and a uniform distribution of faults, simulations show that this probability is about 0.82 for $k=3$, 0.91 for $k=5$, and 0.95 for $k=10$. Error detection is always achieved.

Consider a k -bits Carry-Look-Ahead adder (CLA). Faults in any component (except in the circuit generating the intermediate carry propagated out of the CLA) generate errors that are always corrected, since they are confined within the faulty adder's part. A fault affecting the intermediate-carry generator may propagate the error to the adjacent adder's part possibly preventing error correction: however, the probability of this event is smaller than in then ripple-carry adder, so that the correction probability is very close to 1.

In multiplication, only one operand is partitioned: time-redundant operations and correction are performed as in the error-correcting adder.

RERO can be extended to neural structures such as the direct-mapping architecture shown in Fig. 2. Basic modules are similar to the ones used in error detection; majority voters replace checkers on neuron's outputs and on weighted summations. The arithmetic units in the modules are the error-correcting versions discussed above, but neither error detection nor correction are performed directly at the output of each of them.

In each neuron i , x_k is partitioned in three parts: x_k^L , x_k^I , and x_k^M . The set of interconnection weights w_{ik} is not partitioned. The subsystem computing the linear part is divided the least-, the intermediate-, and the most-significant parts of the neuron.

During the nominal phase, the inner product operation is applied to generate the partial summations $\sigma_{i,k}^L = \sum_{h=1,k} w_{ik} x_k^L$, $\sigma_{i,k}^I = \sum_{h=1,k} w_{ik} x_k^I$, and $\sigma_{i,k}^M = \sum_{h=1,k} w_{ik} x_k^M$ in the least-, the intermediate-, and the most-significant parts of the neuron, respectively; results are stored for comparison. During the first redundant phase, $\sigma_{i,k}^L$, $\sigma_{i,k}^I$, and $\sigma_{i,k}^M$ are produced in the most-, the least-, and the intermediate-significant parts of the neuron, respectively; results are again stored for comparison. During the second redundant phase, $\sigma_{i,k}^L$, $\sigma_{i,k}^I$, and $\sigma_{i,k}^M$ are evaluated in the intermediate-, the most-, and the least-significant parts of the neuron, respectively.

Linearity of the operations allows for propagating any error in the reduced arithmetic units towards the final summation σ_i . Error detection on the summation is always necessary to guarantee the absence of faults inducing errors in any point of the architecture. Avoiding such detection may lead to rely only on the intrinsic neural error masking capabilities. Error detection is achieved by introducing a three input checker: this circuit verifies that the corresponding bits in the three results are identical. This checker contains two registers and a three input comparator. Registers hold results generated during the nominal phase and the first redundant one. The comparator receives these

results and the one generated during the second redundant phase. Hard-wired interconnections rotate results to obtain the same order as the nominal result.

Error correction on the weighted summation is performed by a correcting circuit similar to the checking circuit discussed above: the three-input comparator is replaced by a majority voter.

Interconnection paths are implicitly and fully protected as for simple detection by applying rotation at the neuron's output within the neuron generating the output itself. All faults affecting only one of the three k -bit sections are corrected in the receiving neurons since the other two copies of the received data allow for evaluating the correct weighted summations.

Protection of the activation function $f_i(\cdot)$ is related to the kind of non-linearity, as in the case of detection. If the circuit evaluating $f_i(\cdot)$ is assumed to be hard-core (Fig. 2), correction is applied on weighted summations to avoid the negative influence both of the non-linearities and of the reconvergent paths. Fault location is possible if the fault occurs in the circuits of the weighted summation. For specific implementation of $f_i(\cdot)$, dedicated techniques can be adopted for separate error correction or, at least, detection.

If computation of $f_i(\cdot)$ requires a reasonably small circuit, traditional triplication can be adopted. If the function characteristics do not reduce the correction capabilities, simple error checking is applied to the weighted summation; otherwise, correction is performed by replacing the checker on summation with a voter and by introducing a register holding the corrected weighted summation. A correcting circuit is connected at the output of the three instances of the activation function generator. A register holds the (correct) neuron's output at the voter's output for the receiving neurons during the three phases. Hard-wired rotation is applied to the value stored in such register to produce the neuron output corresponding to the current phase in the receiving neurons.

If the activation function $f_i(\cdot)$ is linear, checking is not required within the individual neuron since the error is always propagated to the output of the activation function. The correcting circuit at the neuron's output is used to provide the correction capabilities.

IV. CONCLUSIONS

Time redundancy with RERO has been proposed to implement fault-tolerant dedicated digital architectures for neural networks, when correctness and reliability of results is mandatory. Error detection is guaranteed, while correction is provided only on statistical basis. The circuit complexity penalty for error detection is less than 5% and quickly decreases as the number of neurons in each layer increases; for error correction, it is slightly higher. The latency for error detection is about twice the nominal one (throughput is halved); for error correction, the latency is three times the nominal latency (throughput is one third of the nominal throughput). Pipelining allows for some improvements.

REFERENCES

- [1] M. Stevenson, R. Winter, B. Widrow, "Sensitivity of feedforward neural networks to weight errors",

IEEE Trans. on Neural Networks, Vol. 1, N. 1, pp. 81-92, March 1990.

- [2] L.G. Chen and T.H. Chen, "Fault-tolerant serial-parallel multiplier", *IEE Proc.-E*, Vol. 138, pp. 276-280, 1991.

- [3] C. Lehmann, F. Blayo, "A VLSI implementation of a generic systolic synaptic building block for neural networks", *Proc. Int'l IFIP Workshop on VLSI for Artificial Intelligence and Neural Networks*, St. Paul de Vence, France, 1990.

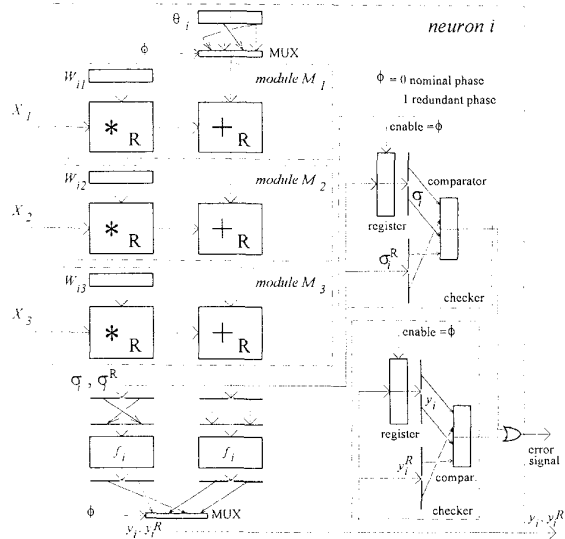


Figure 1: The error-detecting neuron's architecture.

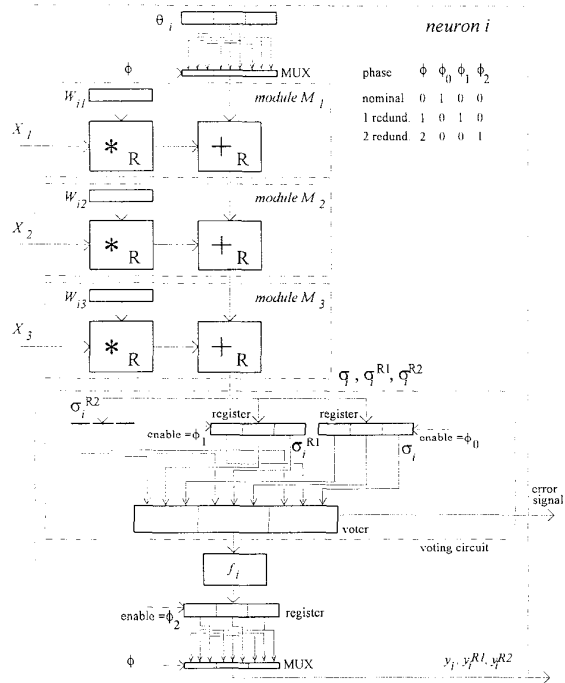


Figure 2: The error-correcting neuron's architecture.