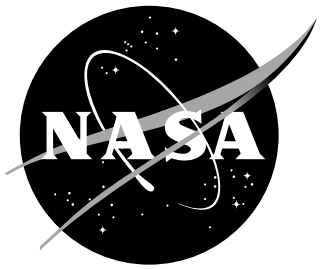
NASA/TM–20220013182



Radiation Tolerance and Mitigation for Neuromorphic Processors

Johann Schumann
KBRWyle, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA

2022

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

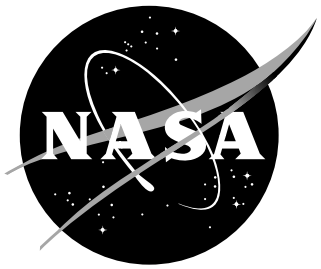
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Help Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199

NASA/TM–2022–0013182



Radiation Tolerance and Mitigation for Neuromorphic Processors

Johann Schumann
KBRWyle, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035

2022

Executive Summary

Neuromorphic processors are designed to execute Deep Neural Networks (DNNs) at very high speed using only a fraction of the electrical power needed to run a DNN on a traditional CPU or GPU. This unique capability makes Neuromorphic processors a prime candidate for space systems, where advanced computational tasks like image analysis, depth map reconstruction, or rover control need to be executed in a power-starved environment.

In contrast to the growing number of applications of Neuromorphic processors in smart phones, the automotive and robotics domain, the space environment is unforgiving because of extreme temperatures and high levels of radiation. Any space system, operating beyond LEO requires computing hardware that is resilient against radiation effects. However, Neuromorphic processors have not yet been designed or tested for their radiation tolerance.

In this report, we consider traditional methods of detection of radiation events and mitigation via redundancy and gauge their effectiveness on DNNs. In contrast to traditional flight software, however, neural networks represent a statistical algorithm, which might affect its resilience against radiation events. We will focus on the analysis of the tolerance of DNNs with respect to radiation events and discuss techniques to detect radiation hits using on-chip triple modular redundancy (TMR) on an Intel Loihi neuromorphic processor and to mitigate radiation damage.

We describe an architecture for on-chip TMR for the Intel Loihi and present results of initial experiments.

Contents

1 Introduction **5**
1.1 Levels of Fault Tolerance and Redundancy 6
1.2 Radiation Tolerance of Software 7
1.3 Radiation Tolerance of DNNs 8

2 Redundancy and Mitigation for Loihi **10**
2.1 The Loihi Architecture and Data Flow 11
2.2 A Simple Redundancy Method 13
2.3 On-Chip Triple Redundancy 15
2.3.1 Architecture 15
2.3.2 Implementation for Loihi 16
2.3.3 Metrics for Error Detection 17
2.3.4 Error Mitigation 18
2.3.5 Failure Injection 18
2.4 Example 18

3 Discussion and Conclusions **21**

List of Figures

1.1 Workflow for introducing redundancy and radiation resilience during network development, training, and deployment 9

2.1 Loihi Hardware architecture. From [2] 12
2.2 Statemachine representation of the basic Loihi execution cycle. From [2] . . 12
2.3 Membrane voltages of two identical neurons with constant input current . . 13
2.4 Twofold on-chip redundancy architecture for MNIST 14
2.5 Comparison of the spiking behavior of two identical groups of 4 spiking neurons 14
2.6 Result of experiment with 2-fold redundancy 15
2.7 Triple on-chip redundancy architecture for Loihi 16
2.8 Triple on-chip redundancy for MNIST on Loihi 19
2.9 Typical MNIST architecture 20

List of Tables

1.1 Techniques for detection and mitigation of radiation failures for hardware, different kinds of software, and NNs 7
1.2 Relevant categories for radiation tolerance for "traditional" software (SW) and neural networks (NN) 8

Chapter 1

Introduction

Neuromorphic processors are designed to execute Deep Neural Networks (DNNs) at very high speed using only a fraction of the electrical power needed to run a DNN on a CPU or GPU. This unique capability makes neuromorphic processors a prime candidate for space systems, where advanced computational tasks like image analysis, depth map reconstruction, or rover control need to be executed in a power-starved environment.

In contrast to the growing number of applications of neuromorphic processors in smart phones, the automotive and robotics domain, the space environment is unforgiving because of extreme temperatures and high levels of radiation. Any space system, operating beyond LEO requires computing hardware that resilient against radiation effects. However, neuromorphic processors have not yet been designed or tested for their radiation tolerance.

There is a large amount of work on the design and analysis of radiation tolerant or hardened computer systems used for military or space systems. Techniques, typically include modular redundancy or the use of processor chips that have been specifically designed for their radiation tolerance (e.g., the RAD750[1]).

Radiation failures are usually caused when high-energy particles of the solar wind or background radiation hit the fabric of a chip at high speeds. Depending on particle type and energy, charges on the chip might change, causing a short temporary wrong outcome of the calculation. Latch-ups have more persistent effects and usually need a reset or reboot to overcome. Finally, radiation particles might cause a permanent damage to the chip or hardware. [1] gives an excellent overview of this topic with a specific focus on neuromorphic hardware.

In all cases, the analysis of radiation tolerance or hardening needs to happen along the following dimensions:

Granularity of the analysis to study the radiation effects: studies can be performed on system level, component level (e.g., the flight software), or on individual software components. Of importance here are, of course, AI components, which are often realized as Neural Networks (NNs).

Type of Software: This plays an important role, as some algorithms might be more tolerant toward radiation events than others. In particular statistical algorithms like NNs can play a specific role here.

Processor architecture: complex computer architectures, which can have multiple CPUs, GPUs, and special purpose ICs (e.g., for neuromorphic processing) need to be studied

[1] `https://en.wikipedia.org/wiki/RAD750`

careful, since different tasks like image preprocessing, NN inference, machine learning, control, or data transfer, are executed on the various components, which might exhibit substantially different tolerance to radiation events. For DNN architectures we typically distinguish between "in-fabric" (i.e., things happening inside the DNN), "on-chip" (e.g., preprocessing, post-processing, routing, control) usually performed by on-chip CPUs and discrete logic, and "off-chip", which includes preprocessing, post-processing, or data/weight transfer. These tasks usually heavily involve the host CPU, external memory, and communication circuits to the neuromorphic chips.

In the course of this analysis, the following questions need to be addressed:

- Where are the "critical paths and components"?
- Can the inherent tolerance/redundancy in the DNN be exploited?
- Which tolerance metrics can be used?
- Which robustification / mitigation techniques can be used?

In this report, we will talk about *fault tolerance*: even if hit: validity of the output doesn't change; hit is not observable from the outside; no specific detection/mitigation is necessary.

Fault detection and mitigation: a hit can be detected reliably and its effects are mitigated using techniques like fast reboot, HW or SW redundancy, or temporal redundancy; the system might experience a short interruption of service, but nominal operations are restored very quickly. A system or component with a suitable fault detection and mitigation technology then exhibits a high radiation fault tolerance.

1.1 Levels of Fault Tolerance and Redundancy

A radiation-hardened system can (or should) consist of several "layers" to provide best levels of fault tolerance (and/or recovery) after a radiation hit. Usually, tolerance is obtained by exploiting redundancy. For a neuromorphic system (running on special hardware or not), we might have:

- hardware redundancy: multiple copies of hardware (identical or different) with voting mechanism
- software redundancy: multiple copies (or variants) of software
- inherent redundancy: use of inherent redundancy and/or robustness of the NN

Table 1.1 summarizes the various techniques for different kinds of components.

In general, detection and mitigation should be transparent to the overall system. Another metric for tolerance/mitigation, which is somewhat orthogonal, is described along these two dimensions:

- spatial redundancy: replication of components are used to avoid interruption of service, potentially located at different locations on spacecraft (or even remote).
- temporal redundancy: if fault occurs during one calculation, the subsequent calculation can be used to detect and potentially mitigate that fault. Here, the hardware/resource overhead is traded toward a slower execution.

Table 1.1: Techniques for detection and mitigation of radiation failures for hardware, different kinds of software, and NNs

Redundancy	Technique	Detection	Mitigation
Hardware	replication	Byzantine voting	selection of good HW, fast reboot
Software	replication	voting/sanity check	restart, recalculation
Software	tolerant by design	N/A	N/A
NN	tolerant (inherent)	N/A	N/A

1.2 Radiation Tolerance of Software

When comparing traditional software (e.g., control SW) and (D)NN execution with respect to radiation tolerance, there are some similarities and striking differences. These differences between "SW" and "NN" should be used to maximize tolerance of the system.

Both SW and NN consist of

- control code (fixed in time)
- constant data
- dynamic data

For SW, the control code is usually much larger, compared to the NN control code, which is essentially just matrix multiplication, addition, and non-linear activation code (ReLU, tanh, etc.).

Constant data for SW are lookup tables, gain tables, etc. of usually smaller size. For NN, constant data are the weights and biases of the NN. These data can be very large.

Dynamic data are variables that change over time. For SW we typically have a state vector as well as intermediate variables. Within NN, the intermediate results of the calculations can be seen as dynamic data. If a NN is trained on-line, then the weights and biases also belong to the dynamic data. Note, that dynamic data do not correspond to "dynamic memory". Here all memory sizes and allocations are fixed.

When executing SW or NN, we can assume that the SW is executed with a certain rate, i.e., the update function (or inference function of the NN) is executed every N ms.

The most striking differences are: (see Table 1.2)

SW for each execution exactly one path through the software is executed. In general, there exist an extremely large number of paths through the SW.

SW only a (small) subset of constant data are accessed (think different tables for different operational or failure modes of the system)

SW in case of a radiation hit on elements of the path, the effects are usually very dramatic: failure of program execution, taking the wrong branch, etc. If an upset occurs on a different path or unused data elements, no effect is noticed.

NN for each execution, the entire NN control code is executed (nested for-loops with arithmetic operations). In principle, there is only one path through the NN control SW, which is executed at the full rate

NN all constant data (weights and biases) are accessed for each iteration.

NN in case of a radiation hit, selected constant values or dynamic data are upset and delivering wrong data. The output of the NN is affected in any case; however, due to the inherent tolerance (see below), the effect can be minimal.

Note that these consideration only consider the effects of an upset with respect to the data. If, e.g., an upset occurs in the digital control areas of the SW or NN, effects can be harsh in any case.

Table 1.2: Relevant categories for radiation tolerance for "traditional" software (SW) and neural networks (NN)

	SW	NN
control code size	large	small
constant data size	small	large-huge
dynamic data size	small	large(?)
code executed	little	all
data accessed	little	all
effect	big	none-big

1.3 Radiation Tolerance of DNNs

As discussed above, neural network algorithms and models can exhibit a substantial inherent radiation tolerance. This allows the designer, in principle to use multiple techniques and strategies to improve radiation tolerance and resilience throughout the entire design and deployment process. Figure 1.1 shows a typical work flow to design a DNN and bring it to execution on a neuromorphic processor.

In an initial step, the DNN architecture is designed for the given task. Here, specific architectures with high redundancy might be used to increase radiation resilience. The training of a DNN is particular important step in the lifecycle. Selection of training paradigms, selection of training and test data sets can play a substantial role in tolerance toward radiation failures.

Once trained, the DNN now can undergo transformation that have the goal to improve stability and resilience. Finally, when the network model is compiled for a specific neuromorphic processor (an Intel Loihi in this figure), hardware-level techniques for detection of radiation failures and their mitigation can be used.

Although techniques can and should be used throughout the entire development and deployment process, in this report we solely focus on the hardware-level redundancy and mitigation techniques for the inter Loihi neuromorphic processor.

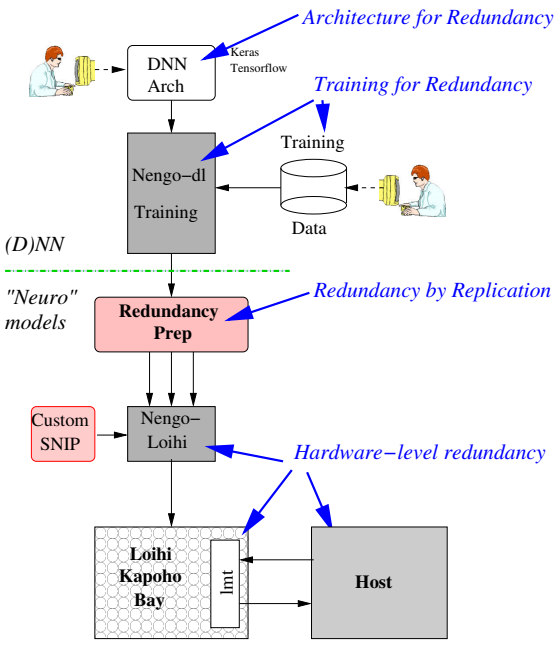


Figure 1.1: Workflow for introducing redundancy and radiation resilience during network development, training, and deployment

Chapter 2

Redundancy and Mitigation for Loihi

For the analysis and design of a redundancy/mitigation architecture for Intel Loihi, a detailed knowledge about the hardware and software architecture is necessary (see Section 2.1 below).

In general, the following redundancy and mitigation techniques are suitable for Loihi. Note that this list is not exhaustive and potentially multiple techniques can be combined to obtain higher levels of resilience.

NN-inherent tolerance: As discussed above, NN architectures and their learning algorithms can be designed in such a way that an improved tolerance against radiation failures can be obtained. In this report, we do not discuss this approach in detail.

On-chip swap: once the fault(s) in the NN have been detected, mechanisms to "repair" the damaged NN need to be used to recover from the problem. In the architecture described in this report, a copy of a known good NN is copied onto the badly behaving one. If both of the NNs are already located on the Loihi, the weights biases, and neuron parameters can be easily copied using the on-chip Lakemont processor. We will describe details below.

The on-chip swap of NNs has several substantial advantages: no lengthy and energy-hungry copying of network data and parameters from and to the host processor is necessary. Therefore, the on-chip swap is fast and highly energy efficient.

Of course, this requires that at least two copies of the NN are present on the Loihi chip(s).

Soft reset: In this mode, a fresh copy of the damaged neural network is copied from the host computer into the Loihi. This operation might take a while as data need to be transferred from the host computer to Loihi.

Loihi reboot: The Loihi and the associated USB circuitry can be reset by the host computer. This means that, after such a reset, the Loihi needs to reboot, which involves setup of the software on the host side, download of the Lakemont boot code, and download of the NN. Although this method is the slowest one, it results in a "fresh" Loihi configuration that should run as expected, unless permanent damage to the Loihi chip has been sustained.

For a redundancy-tolerant architecture with mitigation, several degrees of freedom along the following dimensions exist. Here we briefly discuss those in general.

Redundancy Model: There is a number of basic mechanisms to define redundancy for a NN. The most simple one is pairwise redundancy, where two identical copies of the NN are executed in parallel and their results are compared. This allows us to *detect* single errors in a NN. Other models include: triple redundancy (enabling voting) or multiple redundancy to enable automatic error correction. Obviously, with an increased level of redundancy, the resources (memory, neuron cores, power) increase substantially. Therefore, a more localized redundancy may be defined: only single ("critical") neurons or ensembles are kept redundant to increase the radiation tolerance. The definition of such ensembles can be established with specific analysis and training methods, which can use, e.g., Bayesian techniques, to select ideal candidates for such ensembles.

Checking Metric: The checks to determine, if the redundant (parts of the) NNs are behaving consistently, need to be done often. Therefore, care should be taken to select a suitable metric that meets all needs.

Deciders: once a deviation has been detected, the architecture must produce a suitable result. This can just be an error message, or the result of a voting.

Mitigation: If errors become too severe or occur too often, a mitigation action must be triggered (see above).

The architectures discussed so far are intended for inference-only application, where a pre-trained neural network is being loaded and executed on Loihi. Here, the weights of the NN are not changed during deployment.

On-line training aims to adapt the NN toward proper handling of new incoming data and situations. While such systems exhibit more flexibility and are more powerful in novel environmental situation, their development and deployment requires specific architectural and algorithmic considerations, which will not be discussed here.

Finally, the practical implementation aspects for a redundancy/mitigation architecture needs to be considered. Typical aspects include integration into a development platform (e.g., Nengo, Tensorflow,...), the use of automatic code generation, and use of customized low-level code.

2.1 The Loihi Architecture and Data Flow

For the design of any redundancy architecture, the basic Loihi architecture needs to be taken into account. Figure 2.1 shows a high-level architecture and data flow. The actual neural fabric, consisting of the neuro-cores and the mesh contains the NN (i.e., the synaptic weights, connection structure, and neuron parameters). The highly parallel architecture "executes" the neural network; spikes are sent as messages over the mesh network[1]

Each of the neuron cores contain several memory areas for storage of parameters, synaptic weights, and activation values. For our purpose, we assume that the operations are

[1]For details see `https://www.intel.com/content/www/us/en/research/neuromorphic-computing.html`

executed independently of each other, but SEUs can affect all parts of the neurocores (presumably mainly memory).

The neuro-cores are controlled by 3 on-chip Lakemont processors. They contain a highly simplistic operating system, which

- communicates via serial channels with the host computer,
- sets and monitors the neuro-cores,
- loads and boots the NN definitions, and
- can execute user code

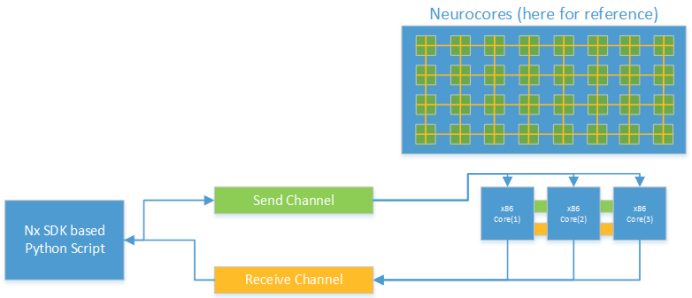


Figure 2.1: Loihi Hardware architecture. From [2]

The overall execution cycle of the Lakemont is split up into three distinct operational phases: spiking, learning, and management. The controlling state-machine is shown in Figure 2.2. A process through these phases comprise the basic "time-step" of the Loihi processor. At each of the steps, custom code can be executed by the Lakemont processors. For our redundancy architecture we exclusively use the "management" phases of the chip to (a) perform checking for consistency, (b) do voting, result and error reporting, and (c) perform mitigation (fast swap) if necessary.

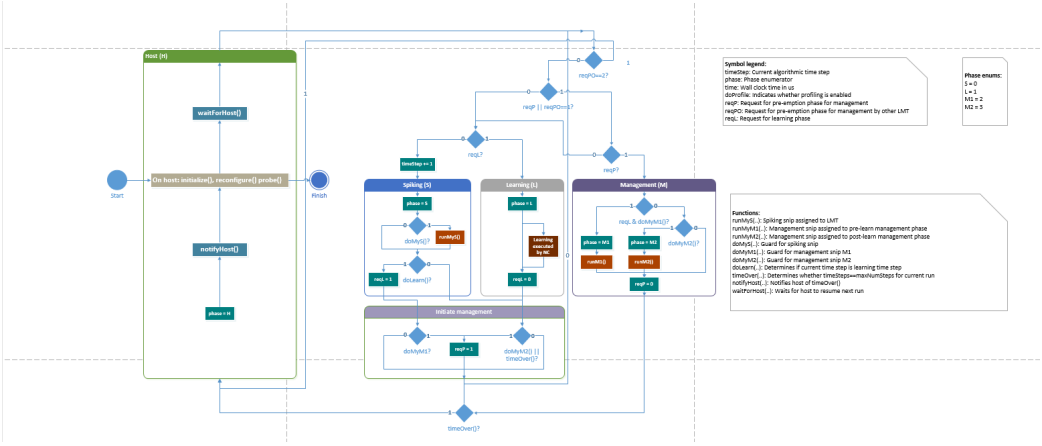


Figure 2.2: Statemachine representation of the basic Loihi execution cycle. From [2]

The registers of the neuro-cores and they local memory are memory-mapped to the Lakemont processors. This enables us to obtain dynamic details of the execution, like

membrane voltage or spike events. This information is usually pulled out using so-called probes.

As an example, consider two identical neurons, which are fed with the same constant input current. Figure 2.3 shows the behavior of the two neurons over time. The orange line, corresponding to the "nominal" neurons shows increases in the membrane voltage, causing the neuron to spike at each 11 time stamps. In a second run (blue lines), an event occurs at $t = 30$ that changes an internal neuron parameter. In the consequence, the spiking frequency of both neurons are now slightly different. However, only after $t = 60$, the difference is large enough to result in different spike times. The bottom panel shows the difference signal.

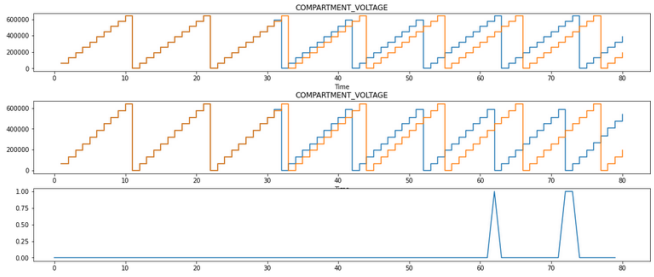


Figure 2.3: Membrane voltages of two identical neurons with constant input current. At $t = 30$, one neuron parameter is changed, leading to a different spike behavior.

2.2 A Simple Redundancy Method

In this section, we describe the detailed architecture of a pairwise redundant NN on Loihi. Figure 2.4 shows the architecture as instantiated for the MNIST example. The interface from/to the host (left-hand side of the figure) is almost exactly the same as for a standard network. The test image is preprocessed, converted into an image of size 28x28. The pixel of each value is sent to the Loihi chip. At the end of each inference cycle, Loihi, however reports back two values instead of one: the probability, with which a specific digit 0,...,9 has been detected, and an error signal.

For this demonstration example, the output of each of the 10 output neurons, as well as 10 error signals are produced at each time step. In a practical implementation, the *maxarg* operation would be carried out on-chip, and only one Boolean error signal would be generated.

The pairwise redundant MNIST architecture in Figure 2.4 is executing the following components on-chip:

- two identical copies of the trained MNIST network. The copies have identical topology, weights, biases, and identical settings of the neurons. (see details below)
- Lakemont 1 is running a customized interface code:
  - it converts the incoming data/spike packets and sends them to the corresponding input neurons of both networks.
  - using the probes mechanism, the spiking status of each of the 10 output neurons for each of the networks are obtained, and a check is performed if the correspond-

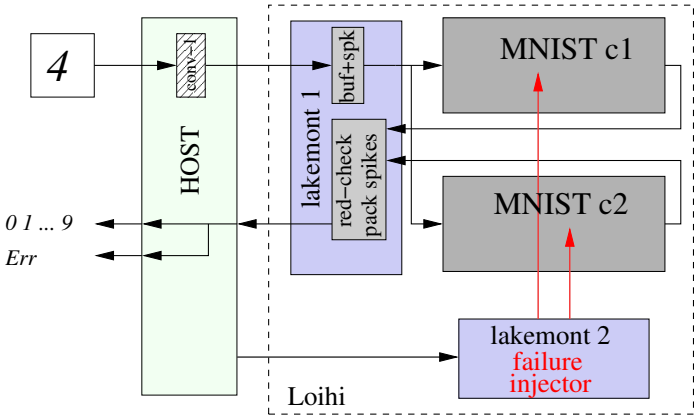


Figure 2.4: Twofold on-chip redundancy architecture for MNIST

ing pairs of neurons behave identical. If not, an error signal for the corresponding pair of output neurons is generated and sent back to the host

The interface code is executed during the management phase of the Loihi cycle (see Figure 2.2.

- (optional) for testing purposes, a failure injection program is being executed on a separate Lakemont processor. It is used to randomly modify weights, change neuron parameters, or induce additional spikes. With that module, radiation effects may be modeled.

Figure 2.5 shows in detail how this architecture works for a group of 4 neurons. The left panel shows the nominal case: the firing behavior of both networks are identical and no error signal is produced.

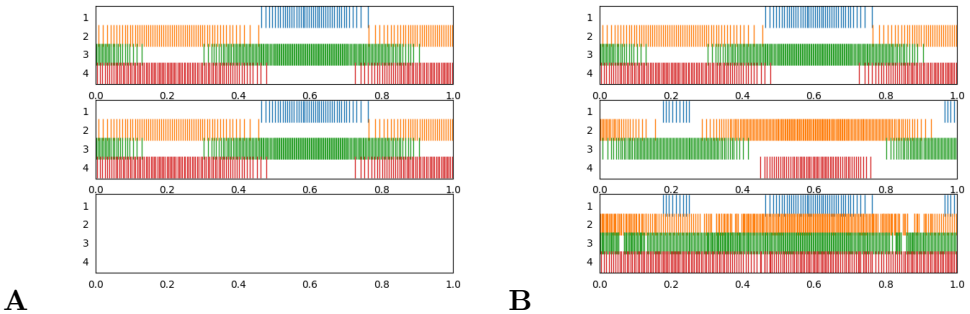


Figure 2.5: Comparison of the spiking behavior of two identical groups of 4 spiking neurons each. At each time-point, the outputs of each of the neurons are compared. Comparison is shown in the bottom panel. **A:** Both groups behave identical in the nominal case. **B:** After "error" injection (modification of one parameter in neuron group 2), the groups are behaving differently and a mismatch signal (bottom) is generated.

Since most of the implementation details for pairwise redundancy are similar or identical to those for the triple-redundancy case, they will be described in Section 2.3.

Figure 2.6 shows a typical result of the pairwise checking architecture on MNIST. The neural networks are running for 500 time steps (the x-axis is normalized to 0,...,1). Each of

the 10 digits are presented 50 consecutive times. The two top panels show th spiking activity (raster plots) for both copies of the MNIST networks. Initially, they are exactly the same, so their spiking behavior is identical and no error signal is produced (3rd panel). At t=0.3 (i.e. at time step 150), a failure is injected; a synaptic weight is changed permanently. The two networks now behave differently and a non-zero error signal is produced. The bottom panel shows the details of the error signal: a direct spike comparison indicates that actually all of the output neurons are affected by that failure.

Although the deviations in spiking of both networks are substantial, in most cases, the actual classification output (i.e., which digit has been recognized) is not affected. This indicates that the checking metric (identical spike timing) might be too restrictive for this classification example (see also Section 2.3.3 for details).

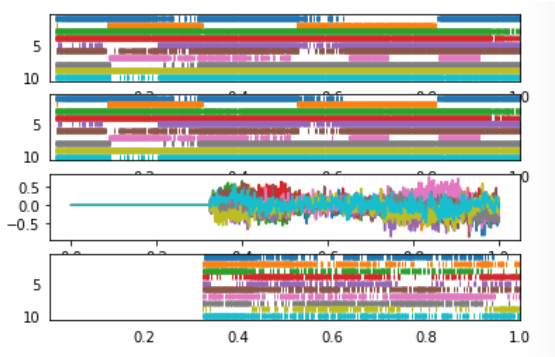


Figure 2.6: Result of experiment with 2-fold redundancy. Error injection happens at t=0.3s. Shortly after that, the error signal (Panel C) shows a non-zero value. A display of the detailed per-output-neuron deviation (panel D) shows that a single injection has affected all outputs neurons in this case.

2.3 On-Chip Triple Redundancy

2.3.1 Architecture

This triple-redundant Loihi architecture with mitigation is based upon the basic mechanisms of the pairwise checking as discussed above. Figure 2.7 shows this architecture, again, instantiated for a trained MNIST network. The major differences which be discussed in detail below, are:

- obviously there are 3 identical copies of the MNIST network
- custom code on one of the Lakemont processors obtain the spiking activities of the output nodes of the three networks and perform a voting. The majority vote is accepted as "the" output, which is sent to the host processor
- the error signal now can have three different values:
  - 0 = all three networks produce the same result (nominal),
  - 1 = two of the 3 NNs produce an identical result; one result is deviating. The voting mechanism returns the majority vote and the error is considered to be corrected

– 2 = each NN is producing a different result. No majority vote can be determined and the error is considered to be uncorrectable

- a hot swapping mitigation procedure: if one of the three NNs is consistently producing bad results, i.e., frequently voted out, this NN is considered to be broken. If a certain threshold is reached, then a copy of one of the "good" NNs replace the broken one in place. Unless the underlying hardware fabric is damaged, this copy now should depend identical to its source and thus should enable valid majority votes.

In the following, we will discuss each of the individual components and their implementation on Loihi.

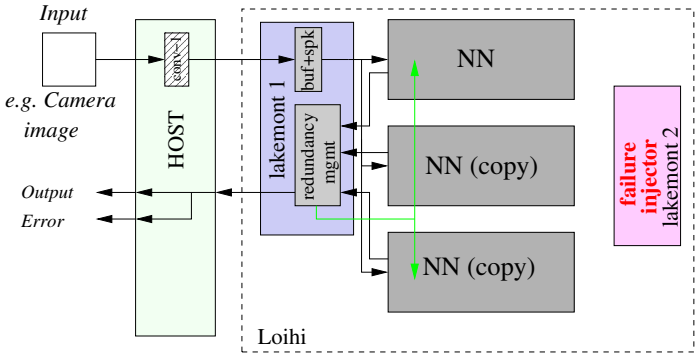


Figure 2.7: Triple on-chip redundancy architecture for Loihi

2.3.2 Implementation for Loihi

The implementation of a TMR for Loihi requires a number of specific preparation steps and customized code is necessary to be executed on the host machine as well as on the Loihi. In principle, our process follows the workflow shown in Figure 1.1. In a first step (not described here), a detailed model for the DNN under consideration is constructed using a toolkit like Keras[2]. Figure 2.9 shows a typical model definition of the MNIST DNN as used as our driving example (Section 2.4). This trained network is then saved to conserve weights and biases.

In order for the DNN to be executed on an intel Loihi processor, it needs to be converted into a Spiking Neural Network (SNN). For this purpose, we use the Nengo[3] framework and Nengo-loihi[4]. For our purposes, Nengo-loihi is of central importance. Nengo-loihi takes an SNN and compiles it into the machine code for the Loihi chip. In addition, it uses automatic code generation to generate customized C code that is executed on the Loihi Lakemont processors. These "SNIPS" are in charge of downloading input from the host machine, injecting spikes into the neural fabric, and obtaining status and output information of selected neurons. This information is then sent back to the host machine.

In order to realize out Loihi TMR architecture, the following additional steps have to be inserted into the workflow:

1. Generation of identical copies of the DNNs. The generation of these copies require that all connections, all weights, and parameters of each neurons are copied. Care

[2] `https://keras.io`
[3] `https://nengo.io`
[4] `https://nengo.io/nengo-loihi`

must be taken that random numbers, which are usually used to preset some of these parameters use exactly the same seed numbers or that any random modification of the DNN is disabled.

2. Generation of "probes" that allow the SNIP on the Lakemont processor to access the activation values and spike counters for each neuron of interest. These probes also generate the code necessary to transmit results back to the host machine. In standard nengo-loihi, probes are used to assemble membrane voltages and spiking behavior only. We "abuse" these probe channels to also carry error signals, status information, and voting results.

3. Extend the nengo-loihi code generation mechanism. The SNIPS, which are generated for execution on the on-chip Lakemont processors must now include code to

   - check the results of the redundant DNNs at each time step and carry out a comparison according to the selected metric,
   - carry out voting to determine a majority vote, which will comprise the result produced by the TMR network,
   - keep statistics on failures and voting results to determine the current quality of each of the redundant networks,
   - code for TMR mitigation, in our case, code to copy the currently best DNN upon the "bad" one, and
   - code for sending results, error signals, and voting information back to the host computer. Our SNIP will also read the relevant ECC information from Loihi's error-correcting memory and report errors.

In our prototype implementation, Python code is used to generate and compile all the artifacts that are then downloaded to Loihi prior to executing the DNN. Storage of the generated SNNs and SNIPS as binaries could eliminate the substantial start-up time.

2.3.3 Metrics for Error Detection

The selection of a suitable metric for failure detection is important, since different tasks might require or suggest different metrics. For example, for a classification task, it is mandatory that all redundant copies of the NN detect the same class. Small deviations in the detection probability can usually be tolerated. On the other hand, a NN-based controller, for which the NN output comprises direct numerical values to the actuators, only a small numerical deviation between the redundant networks should be tolerated.

In principle, the following metrics might be useful:

Spike Count: the redundant NNs are executed in a lock-step style. At each time step, the spike count for each or selected neurons are compared.

Moving average: the use of a moving average (or other filter) can weed out short transient-style deviations if those are acceptable in the system

Argmax: in most classification tasks, the relevant output is calculated as the argmax over the activity of the output neurons. The class label of the neuron with the highest activity is selected.

Spike timing: differences in spike timing might or might not indication a flag-able deviation.

As the checks can be performed on all neurons in the NN, on the output neurons only, or on selected groups, the effort to be spent as well as the granularity can vary substantially. In general, a careful case-by-case consideration will help to decide on the most suitable error metric for the given NN and task.

2.3.4 Error Mitigation

In this architecture, error mitigation is realized by a "hot swap". If one of the three copies of the NN has been tagged by the voting system as "bad/broken", this NN will be overwritten by a copy of the currently best NN. This copy is taking place entirely on-chip and thus is extremely fast. Specifically, the mitigation procedure copies over (a) the synaptic memory, and (b) selected registers of the affected neurocores.

This mitigation process is triggered on user-definable conditions that are evaluated by the voting mechanism. The condition can include the number of discrepancies for each neural network (how many times, a NN has been "voted out"), and frequency and persistence of error condition.

Although this copying procedure can be repeated, care should be taken to a avoid a continuous degradation of NN performance, as source NNs for the mitigation procedure can have failures themselves.

2.3.5 Failure Injection

This architecture also includes a software component (SNIP) for failure injection. This code, which is executed on-chip on one of the Lakemont processors can perform instantaneous or persistent random modifications of the synaptic weight memory and/or synaptic parameters.

Using different probabilistic parameters, this component is helpful to mimic different kinds of radiation events, and to test the redundancy and mitigation architecture.

2.4 Example

Figure 2.8 shows how the failure detection, voting, and mitigation works. For this example, we are using the well-known MNIST network, which, shown a 28x28 image can detect the most likely digit from zero to 9. Consequently the MNIST network has 28x28 inputs and 10 outputs. Figure 2.9 shows a typical architecture.

Figure 2.8A shows the nominal case, when no radiation events happen, B shows the failure case. In both cases, the top Panel 1 shows the output of the architecture (as a raster-plot) over time. The x-axis shows the time in seconds. This output is comprised of the voting result at each time step. The error signals in the other panels are binned into intervals of length 10 time steps. Panel 2 shows the number of recoverable errors (i.e., two of the 3 NNs agree). Since the injected radiation events cause short-term upsets, the total number of errors can go up and down. In the nominal case (A), no errors occur. Panel 3 shows the number of unrecoverable errors, i.e., all the NNs disagree. In the injection case, the first unrecoverable error only occurs around 0.18, despite the fact that numerous radiation events occur, as indicated by the large number of recoverable errors (Panel 2). A bit later (t=0.2s) two unrecoverable errors occur shortly one after the other, causing a

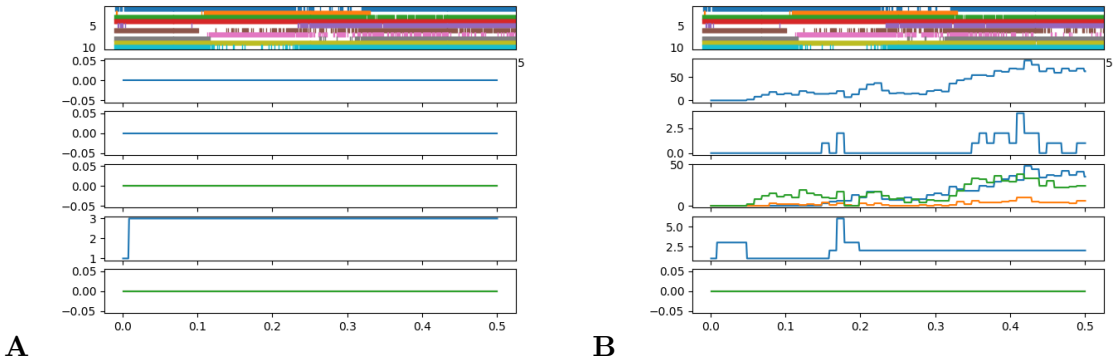


Figure 2.8: Triple on-chip redundancy for MNIST on Loihi

mitigation step. This immediately brings down the number of unrecoverable errors again for a long time. Only later, after about t=0.35s the number of errors increase substantially. Since in this example, only one mitigation step is allowed, no more mitigation corrections can be done and the system behavior deteriorates.

In such a case, a second mitigation step might be tried, or a soft reboot and restart of Loihi with fresh copies if the NNs can be attempted to consolidate this situation.

Panel 3 shows the number of errors over time for each of the three NNs. The differences in behavior depend on the random failure injection. It can be seen clearly that the "green" NN starts performing badly at around t=0.1s. When the mitigation is triggered by the second set of non-recoverable errors, the "green" NN, as the worst-performing on is overwritten. It can clearly be seen that the error rate drops sharply directly after the mitigation.

Panel 4 shows, as internal information, which of the NN provide their output. In the nominal case, always NN 3 ("green") is selected, due to the implementation of the voter. When errors start to show up in NN 3 (t=0.05s), NN 1 wins the votes. The spike at t=0.18s is technical only and shows that a mitigation step has happened. Afterwards, the newly resurrected NN 3 wins the votes, before NN 2 ("orange") takes over. The bottom plot shows the number of ECC errors in the Loihi synaptic memory. In both cases, this number remains at zero, because ECC errors cannot be injected.

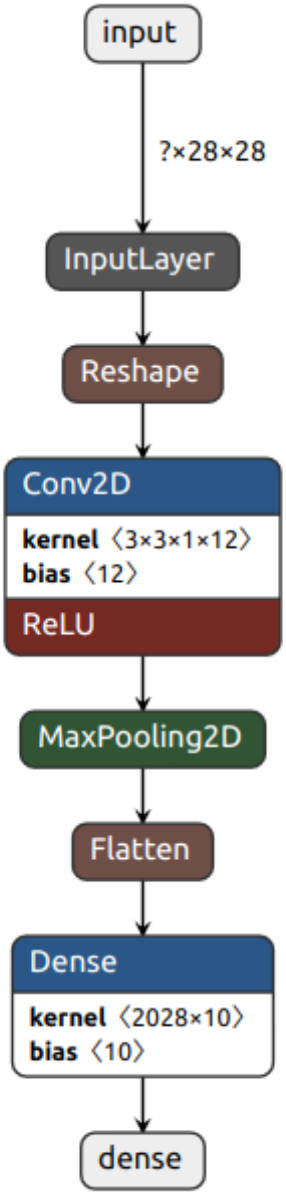


Figure 2.9: Typical MNIST architecture

Chapter 3

Discussion and Conclusions

In this report we have described an architecture to realize an energy efficient redundancy and mitigation mechanism for intel Loihi. Two or three identical copies of the DNN under consideration are executed in parallel on the Loihi chip. Customized, auto-generated C code (SNIP) is executed on the on-chip Lakemont processors to handle the communication with the host processor, to compare the results of the individual DNNs, to vote on the result, and, in case of repeated failures of on DNN, perform a hot-swapping by replacing the broken DNN with a copy of the currently best one. We demonstrated this architecture using the well-known MNIST digit classification example.

The architecture described in this report should only be seen as an initial step toward redundant architectures to detect and mitigate the effect of radiation events. There is a number of important points for future work ranging from fundamental issues to improvement of the actual implementation/generation.

Acknowledgements

Keerthana Kannan and Michael Furlong helped substantially on various aspects of principles and implementation of the Loihi TMR.

Bibliography

1. Richard Alena. Mission radiation environment modeling and analysis: Avionics trade study for gcd rad-neuro project. Technical Report NASA/TM-2022001177, NASA, 2022.

2. Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A. Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R. Risbud. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934, 2021.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01--2022	Technical Memorandum	

4. TITLE AND SUBTITLE
Radiation Tolerance and Mitigation for Neuromorphic Processors

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
RadNeuro

6. AUTHOR(S)
Johann Schumann,

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Ames Research Center, Moffett Field, CA 94035

8. PERFORMING ORGANIZATION REPORT NUMBER
L–

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–20220013182

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT

Neuromorphic processors are designed to execute Deep Neural Networks (DNNs) at very high speed using only a fraction of the electrical power needed to run a DNN on a traditional CPU or GPU. This unique capability makes neuromorphic processors a prime candidate for space systems, where advanced computational tasks like image analysis, depth map reconstruction, or rover control need to be executed in a power-starved environment.
In contrast to the growing number of applications of neuromorphic processors in smart phones, the automotive and robotics domain, the space environment is unforgiving because of extreme temperatures and high levels of radiation. Any space system, operating beyond LEO requires computing hardware that is resilient against radiation effects. However, neuromorphic processors have not yet been designed or tested for their radiation tolerance.
In this report, we consider traditional methods of detection of radiation events and mitigation via redundancy and gauge their effectiveness on DNNs. In contrast to traditional flight software, however, neural networks represent a statistical algorithm, which might affect its resilience against radiation events. We will focus on the analysis of the tolerance of DNNs with respect to radiation events and discuss techniques to detect radiation hits using on-chip triple modular redundancy (TMR) on an Intel Loihi neuromorphic processor and to mitigate radiation damage. We describe an architecture for on-chip TMR for the Intel Loihi and present results of initial experiments.

15. SUBJECT TERMS
Neural Networks, Neuromorphic Processor, Hardware, Radiation Tolerance

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Information Desk (email: help@sti.nasa.gov)
U	U	U	UU		19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18