MigSpike: A Migration Based Algorithms and Architecture for Scalable Robust Neuromorphic Systems

Khanh N. Dang, Nguyen Anh Vu Doan, and Abderazek Ben Abdallah, *Senior Member, IEEE,*

Abstract—While conventional hardware neuromorphic systems usually consist of multiple clusters of neurons that communicate via an interconnect infrastructure, scaling up them confronts the reliability issue when faults in the neuron circuits and synaptic weight memories can cause faulty outputs. This work presents a method named *MigSpike* that allows placing spare neurons for repairing with the support of enhanced migrating methods and the built-in hardware architecture for migrating neurons between nodes (clusters of neurons). *MigSpike* architecture supports migrating the unmapped neurons from their nodes to suitable ones within the system by creating chains of migrations. Furthermore, a max-flow min-cut adaptation and a genetic algorithm approach are presented to solve the aforementioned problem. The evaluation results show that the proposed methods support recovery up to 100% of spare neurons. While the max-flow min-cut adaption can execute milliseconds, the genetic algorithm can help reduce the migration cost with a graceful degradation on communication cost. With a system of 256 neurons per node and a 20% fault rate, our approach minimizes the migration cost from remapping by 10.19× and 96.13× under Networks-on-Chip of 4×4 (smallest) and $16 \times 16 \times 16$ (largest), respectively. The Mean-Time-to-Failure evaluation also shows an approximate 10× of lifetime expectancy by having a 20% spare rate.

Index Terms—Fault-tolerance, Spiking Neural Network, Neuromorphic System, Network-on-Chip, Max Flow, Migration.

1 Introduction

Brain-inspired computing or neuromorphic computing is the next generation of artificial intelligence to extend to areas of human-like cognition. Spiking neural network (SNN) [1], which is considered the third generation of Neural Network, is a novel model for arranging the replicated neurons to emulate natural neural networks existing in biological brains. The computational building blocks are the replicated version of neurons that receive, process, and send possible output spikes. Recently, several works have been done to integrate a large number of neurons on a single chip while providing efficient and accurate learning [2], [3], [4], [5], [6], [7], [8], [9]. While software implementations of SNNs [7], [8] have demonstrated the ability to emulate the operation of biological brains, several works [2], [3], [4], [5], [6] on implementing SNNs hardware have been proposed to accelerate the performance and to gain energy and memory efficiency. Conventionally, the system usually consists of multiple nodes (or clusters) of neurons connected via an on-chip communication infrastructure [3], [9]. Expansion using a multi-chip system and off-chip interconnects is also a viable solution for scaling up SNNs [2], [3].

By integrating a large-scale spiking neural network, we also encounter the reliability issue due to the accumulating probability of faults [10]. Mathematically, the possibility of having defective neurons and memory blocks is increased when we upscale the system. Naturally, an SNN is resilient against spike and weight faults; however, accumulating the faults over a long period could lead to undesirable inaccuracy outputs.

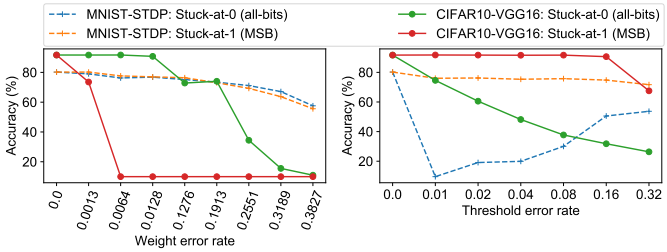


Fig. 1: Impact of faults on accuracy of MNIST dataset with unsupervised STDP based SNN size 784:100 and CIFAR-10 with spiking VGG-16.

To understand the impact on SNN, we randomly inserted several faults and tested with the test images in the MNIST and CIFAR10 datasets. For the MNIST dataset, the SNN model is 784:100 with lateral inhibitory connections adopted from [11] and run on BindsNet [8] simulator. This network follows the winner-take-all principle, where a firing neuron inhibits other neurons. The weights are pre-trained using STDP as in [11]. For the CIFAR10 dataset, we adopt the VGG-16 model with the conversion in [12].

- *Khanh N. Dang is with both the VNU Key Laboratory for Smart Integrated Systems (SISLAB), VNU University of Engineering and Technology, Vietnam National University, Hanoi, Hanoi 123106, Vietnam and with Adaptive Systems Laboratory, Graduate School of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, Japan. E-mail: khanh.n.dang@vnu.edu.vn; khanh@u-aizu.ac.jp (Corresponding authors: Khanh N. Dang)*
- *Nguyen Anh Vu Doan is with the Chair of Integrated Systems, Technical University of Munich, 80333 Munich, Germany E-mail: anhvu.doan@tum.de.*
- *Abderazek Ben Abdallah is with Adaptive Systems Laboratory, Graduate School of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, Japan. E-mail: benab@u-aizu.ac.jp*

Figure 1 illustrates the accuracy drop when inserting faults into the weight SRAM and thresholds. Once we inserted stuck-at-0 or stuck-at-1 faults into the weight memory of the MNIST model, as shown in Figure 1, we can notice that the accuracy drops are ineligible for a small number of faulty weights thanks to the natural fault-resilience of SNNs. However, when the number of faults increases, the accuracy starts dropping significantly. We can observe a similar behavior with stuck at-0 in VGG-16 for CIFAR10. However, the stuck-at-1 is critical for VGG-16 as higher weight leads to the domination of some features in the convolution. Once the error rate reaches above 0.0064, the accuracy stuck at 0.1, which means the VGG-16 only outputs one label. On the other hand, a defect in a computation unit is more critical in the MNIST model. A single stuck-at-0 (which is 1% in 100-neurons) on a threshold register can easily make a constant firing neuron, which drops the accuracy significantly to around 10%. This is equal to assigning one label for all testing images of MNIST. Once two or more neurons have stuck-at-0 faults at their threshold registers, two or more neurons start to compete, which increases the overall accuracy; however, the accuracy is still much lower than the non-faulty results. On VGG-16, the stuck-at-0 is a critical problem. The spike can explain this is generated by comparing the membrane potential with a threshold. As the membrane potential accumulates weighted spikes, stuck-at-0 in threshold has similar behaviors as stuck-at-1 in weights. While stuck-at-1 seems not so critical as the accuracy maintained with the error rate in the threshold lesser than 0.16, stuck at-0 in thresholds constantly reduces overall accuracy.

In summary, although SNNs are resilient against faults in some certain levels, as we demonstrated, protecting them is still necessary for critical applications or lifetime reliability where the number of faults can be accumulated. To help recover from neuron failures, adding redundant neurons for replacements can be a viable solution. However, the following problems should be addressed:

1) First, adding redundancies to have spare modules to correct can be helpful but not always optimal. A clustering fault distribution [13] could make some nodes not repairable, and other nodes even have unused spare modules.
2) Second, it lacks algorithmic methods to protect the SNN at the system level. Mapping for multi-core systems is already challenging and can be more difficult with faulty cores and communication constraints [14].
3) Third, even if we can adopt a mapping algorithm from multi-core systems [14], its complexity is exceptionally critical. For instance, heuristic search complexity is factorial, and Integer Linear Programming is an NP-complete problem.
4) Fourth, most of the existing mapping algorithms for multi-core systems consider the communication cost as their major goal [14]. Meanwhile, mapping for SNNs mainly considers the connection on the crossbar or the distortion on spike intervals. However, as Neural Networks usually have a massive amount of parameters (i.e., AlexNet [15] has 60 million parameters and 650,000 neurons), the migrating time of the recovery task should also be considered. Downloading the whole parameters could be challenging with low-power nodes.

As motivated by the above challenges, this work proposes a method named *MigSpike* to deal with both fine-grain recoveries at node-level and across nodes at the system-level. First, we use redundant nodes and neurons to deal with the failures of neurons. The unbalancing fault rates can be solved by using system level migration to move faulty neurons to different nodes. Moreover, we take the migration cost as our system's target to reduce the number of updated weights and parameters for recovery. Our method can reduce the migration cost by conserving the existing memory of neurons in the system and only moving the migrating neurons. Our contributions are summarized as follows:

- A design of Network-on-Chip based Spiking Neural Network based on a neuron migration approach. With the support of AER (Address Event Representation) and node address Look-Up Tables, the system can remap a neuron to another.
- A node-level recovery mechanism for neuron failures by placing spare neurons into each node (neuron's cluster) and a system-level recovery mechanism for neuron failures when the node-level approach fails to correct faulty neurons. Here, the faulty neurons are migrated by M hops from their origin to reduce the migration cost.
- A max-flow min-cut using Ford-Fulkerson implementation to solve the graph-flow provides a mapping method for defective neurons. By lowering the M to 1, the Ford-Fulkerson implementation's time complexity is polynomial as $O(N^3)$ (N: number of nodes).
- An augmented algorithm to deal with the unsolvable problems of $M = 1$ hop is also solved by recursively increasing the migration distance (M).
- A genetic algorithm (GA) approach to solve the mapping problem to optimize the migration costs in conjunction with other cost functions. We here design the new crossover and two mutation functions to allow the solutions to evolve and converge.

This paper is organized as follows: Section 2 presents the existing literature on protecting neuromorphic systems. Section 3 gives the baseline SNN architecture to provide an overview of the proposed platform. Section 4 presents our proposal on the neuromorphic mapping method using a graph-based algorithm. Section 5 evaluates the proposed methods and finally Section 6 concludes this work.

2 Related Works

This section summarizes the related works on protecting neuromorphic systems in three significant aspects: communication, computation, and memory. Moreover, mapping methods for SNN to recover from faults are also summarized.

2.1 Memory Protection

Since memories are vulnerable to permanent and transient faults, protecting them is needed for highly reliable systems. One of the most popular methods is to use Error Correction Codes, such as Hamming or its extended version [16], which can correct one flipped bit in the *codeword*. For multiple bits upset, multi-bit correction such as Orthogonal Latin Square Code [17] or Triple Adjacent Error Correction [18] can be used. Another recovery method for memory is to add a spare row or column and use the spare one as a replacement for the faulty one [19].

On the other hand, memory errors can be tolerated in neural network applications by accepting a specific loss of accuracy. As analyzed in [20], a CNN application lost 5.7% in terms of accuracy with an error rate of 0.0065. Our analysis in Figure 1 also shows an acceptable loss while inserting a similar error rate. In summary, we can either protect the memory using error correction code or accept accuracy loss under a certain noise level.

2.2 Communication Protection

Since spikes, neurons' parameters, or weights could be transmitted within the system or external memories, corruption in these values could lead to inaccurate results. Therefore, protecting their integrity is an important problem. Apparently, inheriting Error Correction Codes [16], [17], [18] from memory protection could be helpful. Here, the data is protected under a certain number of flipped bits.

Another type of error in communication is the misrouting or arbitration failures [21]. In these cases, recovery using an alternative routing path or redundancy could be used. By avoiding the failure point and providing a viable routing, a fault-tolerant routing algorithm [10], [22] can help overcome these types of errors. On the other hand, by providing redundant modules [23], the system can replace a faulty module with a healthy one for recovery.

2.3 Computation Protection

Faults in a computation module could be critical to SNNs, as we previously demonstrated. Therefore, protecting computation units is substantially vital. In [24], the authors proposed a method to protect the systolic array by bypassing and retraining. By pruning the faulty part of computation and retraining the model, the system can accept a certain fault level. *Johnson et al.* [25] also presented a method to retune the spiking model with variable thresholds and operating frequencies for allowing fault tolerance. A traditional method such as N-modular redundancies with a majority voting [26] could also be used to ensure the correctness in this case. However, it leads to high area costs and power consumption.

As large-scale SNN systems usually utilize Network-on-Chip as the communication infrastructure, the computation protection method can use extra cores and remapping algorithms. The fault-tolerance NoC system with homogeneous cores can be solved by using Integer Linear Programming (ILP) as in [14], where the authors tried to optimize the communication cost (summary of the traveling distances). However, the ILP problem is NP-complete, which cannot deal with larger scales. The works in [14], [27] also present a Particle Swarm Optimisation (PSO) solution to reduce the complexity of the mapping algorithm. Although the PSO-based approach can significantly reduce the run-time, it cannot guarantee the optimized result.
Moreover, PSO has a high space complexity for storing all particles. For reliability aware mapping, *Namazi et al.* [28] presented an approach to map tasks to homogeneous NoC architecture using a Mixed Non-Linear Programming model. Despite providing promising results, the mentioned approaches only target conventional multi-core systems. For our large-scale SNN system, since each node can have multiple computing units itself, internal node recovery is also possible instead of requiring external spare cores. Also, the recovery methods do not take the migrating time between cores into account.

2.4 SNN Mapping

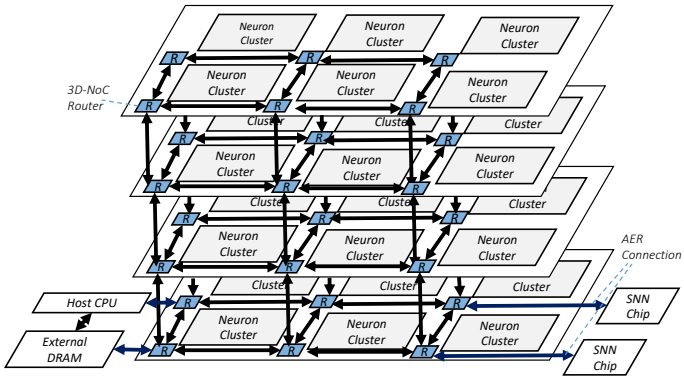
Since the targeted system is an NoC-based multi-cores one, we can use both the SNN mapping methods [3], [29], [30], [31], [32], [33] and conventional multi-core NoC mapping methods [14] for placing neurons. While the conventional multi-core mapping solutions such as ILP or PSO prove their efficiency, mapping for NN is highly complicated due to a large number of neurons. The conventional SNN mapping method has two phases [32]: (1) Partitioning: cluster the NN into groups of neurons; (2) Mapping: map the groups of neurons to hardware. However, there are some problems: (1) both graph partitioning and mapping are NP-hard, which might not be solved optimally in polynomial time; therefore, a non-optimal solution can be justified; (2)the layered SNN applications have the node as the layers itself; and (3) the conventional methods do not take into account the multi-casting manner in communication. Lagrange multipliers [29] can reduce the run-time complexity; however, we still observe the long execution time. Since mapping for each neuron is not feasible, partitioning then the mapping is a potential approach [30]. We have to note that partitioning is an NP-hard problem. In [32], the authors adopted the Kernighan-Lin (KL) partitioning method for reducing the complexity despite not providing optimal results.

3 Baseline Neuromorphic System

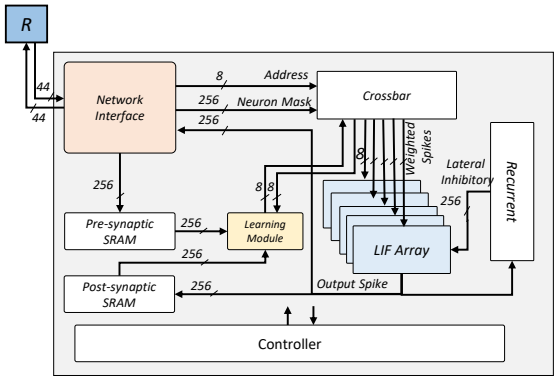
This section presents our baseline SNN architecture [34]. We first give an overview of the SNN architecture. Then, the communication infrastructure is shown as the backbone of the inter-neural transmission. We then briefly present the design of the cluster (or node) and the neuron itself.

3.1 SNN Architecture

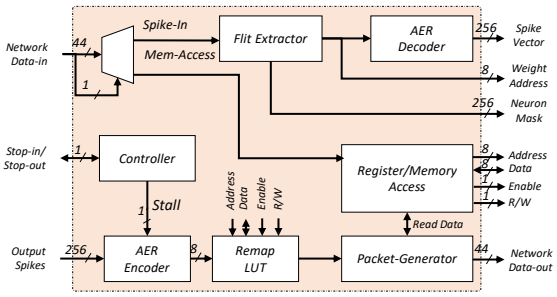
The overall architecture of our neuromorphic system is shown in Figure 2(a), which is designed on a 3D-IC (Three Dimensional Integrated Circuits) infrastructure to model the three-dimensional structure of the brain. Here, the neurons and their synapses are clustered in neuron clusters or nodes. Instead of using point to point connection between neurons in a biological brain, we use a Network-on-Chip infrastructure to support communication. The 3D-Mesh topology is used where each node is attached with a



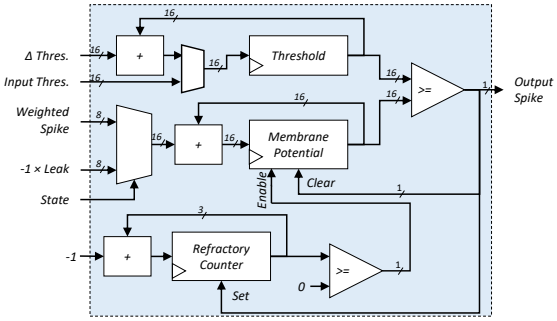
(a) System block diagram.



(b) Block diagram of a neuromorphic node (neuron cluster).



(c) Network Interface architecture.



(d) LIF neuron architecture.

Fig. 2: Baseline SNN architecture.

router of six communicating directions. Combining with the packet-switching mechanism, we can replicate the communication between neurons via synapses. While a 3D-Mesh NoC handles the communication, the computation is done by a neuron cluster (node) where its architecture is shown in Figure 2(b). Here, the controller manages the computation by time step, where the synchronization is done via NoC communication. The incoming spikes in AER (Address-Event-Representation) protocol are stored in memory and decoded to obtain the address and the read enable signal for the weight memory. By taking the weights from memory, the system emulates the weighed spikes for LIF neuron inputs.

3.2 Inter-neural Communication

The inter-neural interconnect consists of multiple routers (R) to handle the communication between the neuron clusters [34]. Here, to support the 3D-Mesh topology, we adopt our present works on 3D-NoC [35]. The adopted 3D-NoC supports several fault-tolerance features on input buffers, crossbar, fault-tolerant routing, transient faults in routing logic and data. We embrace the unicast-based multicast from our previous work in [36]. The packet in the inter-neural is a single flit where two types of flit are supported. The first type is the spike between neurons in AER format. The AER format flit is converted to the address of the weight SRAM to feed to the SRAM. The second type of flit is memory access. To read and write the memory cells and registers in the neuron cluster, a flit should provide the instruction and the required argument (address). Here, the memory access flits are issued by a master (or external host) processor in the system. We support two types of read/write commands: single and burst. The individual read/write only provides access to one element per request, while an argument of length must follow the burst ones. The Network Interface (NI) converts the requested address to the local address at each weight memory or LIF array. Figure 2(c) shows the block diagram of the Network Interface. The input spikes are categorized into either input spikes or memory accesses. With the memory accesses, the NI provides an interface to read and write the data in all registers and memory blocks of the node. The read instruction makes the NI returns the master processor the value of the requested address. With the input spike from the network, the NI decode phase gets the weight SRAM address and feeds it to the weight memory. For multi-layer SNNs or sparsity connections, the *Flit Extractor* provides the read-enable signal for different layers or different links, which are used in the weight memory. As a result, a node can have multiple AERs at the same address but for other neurons. The LIF array's output spike is fed into the AER decoder, which extracts the address of bit one (firing neuron). This address is then serially sent to the remap Look-Up-Table (LUT) to obtain the AER value in the receiving nodes.

3.3 Spiking Neuro-processing Core (node)

After receiving the address of the corresponding weight and the enable signal, the series of weighted inputs will be sent to the dedicated LIF neuron, which accumulates the value, subtracts the leak, and check the firing condition. The output spike is stored in a post-synaptic SRAM and sent to the Network-on-Chip. The computation of the neuron is done with our hardware LIF array. In this work, we select Leaky-

Integrate-and-Fire as the neuron model. We have to note that most hardware-friendly neural architecture focuses on either Leaky-Integrate-and-Fire (LIF) or Integrate-and-Fire (IF) model due to their simplicity. Figure 2(d) shows the architecture of a LIF neuron. The weighted input (i_wspike) is fed into an adder+register structure to accumulate the value. At the end of each time step, the leak's inverted value is fed to reduce the membrane potential. The membrane potential is compared with the threshold to check the firing condition. If the neuron fire sets the count down $refac\_cnt$ to keep the neuron stops working for several time steps (as refractory).

4 Proposed Recovery Method for SNNs

In this section, we first formulate the problem of remapping the faulty neuromorphic system. We then present the proposed algorithm for migrating the unmapped neurons.

4.1 Problem Formulation

Here, we assume that the working system S has N nodes (or neuron clusters) where each node has E_i ($i = 0, 1, \ldots, N-1$) neurons. In serial systems [3], [37], the number of neurons is equivalent to the number of memory slots a node can stores. Note that the value of E can vary up to design and can be different between clusters in heterogeneous systems. In short, the total number of neurons in the system S is $X = \sum_{i=0}^{N-1} E_i$. Here, we also assume that the desired SNN application requires W neurons. A possible application must have $W \leq X$.

4.1.1 Fault-tolerance

Because we support fault-tolerance in our system architecture, we consider R spare neurons, which $R = X - W$ as the repairing source. Once $k \leq R$ neurons are faulty and must be removed from the system, our problem formulation is to remap these k neurons to R spare neurons. If $k > R$, the system S cannot correct, and an off-chip migration should be considered (i.e., plugging a new chip and migrate to it). If the system can remap the function of k neurons, parameters, and weight of k neurons, we archive $k-$fault tolerance. In term of repair-ability, we divide it into two levels:

- *Node-level recovery*: If the node has enough spare neurons to correct its failed ones, it corrects internally by remapping. If it fails, the *system-level recovery* is used. A copy of the weights and parameters stored externally is read and written to the spare neuron.
- *System-level recovery*: If there are not enough spare neurons in a node for its internal recovery, the migration of neurons happens across nodes of the system. A migrating neuron can move from its original node to a new one. The corresponding weight, mapping LUT elements, and neuron status are copied to the new neuron. If a node happens to have more neurons to be mapped than E_i as designed, the unmapped neurons will migrate.

In *node-level recovery*, the number of spare neurons in each cluster is an important parameter. Here, we assume the fault rates of all clusters are similar; therefore, we will plan to add the same number of spare neurons in each node. Non-uniform fault rates (for example, central nodes have more faults) will need *system-level recovery* more frequently as the nodes lack spares for recovery. On the other hand, if the fault-rate can be predicted, we can distribute the spares based on the predicted fault rates.

In *system-level recovery*, if the communication is guaranteed as reliable, the system must support up to R fault-tolerance ($k = R$). Figure 3 shows the system model of S with $N = 9$ nodes of $E_i = 256$ neurons ($X = 2,304$) and a possible solution. The system requires $W = 2000$ neurons to perform the application and maps the neuron uniformly, as shown in Figure 3(a). In this mapping example, there are 33 or 34 spares neurons per node, which allows the node to correct up to 33 or 34 faulty neurons. Figure 3(b) illustrates the case of node (0,0) has 10 defective neurons and they are internally corrected using *node-level recovery*. However, Figure 3(c) shows the case of 100 defective neurons in the node (0, 0), which it fails to recover using *node-level* repair. Figure 3(d) shows a mapping flow that maps the faulty neuron to the node (0,0), (0,1) and (1,0), which are the current node and its neighbors. In serial systems [3], [37], designers might need to treat memory units and computation units independently. For memory units, it is similar to the above formulation. For faults in computing units, as the physical neuron can be one per node, adding redundancies is in memory and biological neurons.

4.1.2 Remapping problem

One of the major issues is how to remap the SNN to recover from faulty neurons. Traditionally, one of the optimization goals for remapping is to minimize the following communication cost [14]:

$$F_{cost} = \sum_{i=0,j=0}^{W} d_{ij} \times c_{ij} \tag{1}$$

where d_{ij} and c_{ij} are the distance and the connection status between node i and j. If we use c_{ij} as binary $(0/1)$, F_{cost} is the sum of traveling distance between neurons. With this kind of optimization, we only need to rerun the mapping algorithm with faulty information. However, in large-scale systems, migrating neurons require an enormous amount of memory access. Therefore, this work optimizes the migration cost, which is the cost of migrating neurons of the new mapping method:

$$M_{cost} = \sum_{i=0,j=0}^{W} d_{ij} \times m_{ij} \tag{2}$$

where m_{ij} is the number of migrating neurons between node i and j. Since the data (weight memory, threshold, etc..) within the faulty neurons can be corrupted, the system should write back from its host CPU. The moving distance is d_{0j}, where node 0 is the node attach to I/O module. This optimization considers high availability for the system where it needs as least as repairing time as possible.

4.2 Proposed Max-Flow Min-cut based Algorithm

In this part, we present our proposed algorithm to enhance the reliability of the SNN system. Our main target is to

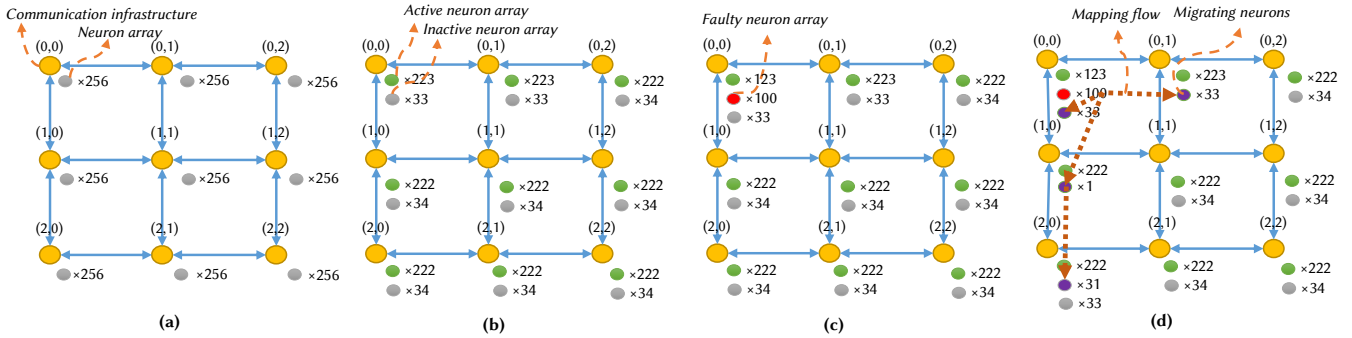


Fig. 3: System model for fault tolerance SNN: (a) Designed SNN system using nodes of neurons with an initial mapping; (b) *node-level recovery*; (c) The case *node-level recovery* fails to correct; (d) *System-level recovery*: a mapping flow of 100 faulty neurons to its node's neighbors. Values next the circle indicate the number of neurons in the circle type (gray: healthy and utilized; gray: healthy and spared; red: faulty; purple: migrating).

optimize the M_{cost} in Equation 2. We first present the max-flow min-cut theorem for the optimal flow. Then, the augmented versions multi-layers design and tackling the limitation of the max-flow min-cut theorem are discussed.

4.2.1 Max-flow min-cut theorem

One of the most common methods to find a flow between sink and source in a graph model is to use the max-flow min-cut theorem to optimize it. Here, we use the same principle: the sources are the faulty neurons, and the sinks are the spare ones. However, the multi-sink multi-source problem is usually complicated and can be converted to the conventional one using a virtual sink and a virtual source.

One of the main reasons to choose the max-flow min-cut approach as the solution for the remapping problem is a good trade-off between efficiency and execution time (or memory footprint). Compared to a greedy search approach from faulty neurons to spare neurons (evaluated later), the max-flow min-cut method provides a better mapping distance by creating a flow (chain of migrating). Meanwhile, the max-flow min-cut complexity is smaller than meta-heuristic methods (i.e., Genetic Algorithm or Particle Swarm Optimization). Moreover, these meta-heuristic methods require a huge memory footprint, which might not be optimal for a low-cost host CPU. Remapping the whole system by reusing the mapping method is a viable solution; however, as we previously discussed, the migration cost can be high due to the tremendous amount of memory transactions needed.

Figure 4 shows the flow graph for the fault tolerance in Figure 3(c). To support moving neurons, we firstly build a virtual source and virtual sink for the flow graph. Then, the connection between the virtual source to the faulty node has a capacity as the number of faulty neurons (i.e., 100 in Figure 4(a))). From the connected node, the flow capacity to its neighbors is the number of healthy neurons in the neighbor. For instance, Figure 4(a) shows the capacity of the flow between (0,0) and (0,1) is 256 since the node (0,1) has 256 neurons and all can be migrated. For each node, there is a virtual flow to a virtual sink with the capacity of the number of spare neurons that are available to be used. For instance, Figure 4(a) shows the capacity of the flow between (0,0) and t is 33 since the node (0,0) has 33 spare neurons. As we can realize in Figure 4, the flow only comes from one node to one of its neighbors as we limit the traveling distance of migration to 1. In other words, Equation 2 has $d_{ij} \leq 1$, which can reduce the migration cost. After solving using a max-flow min-cut solution, we end up having a flow map in Figure 4(b). Here, we can convert back to the NoC-based SNN to have the new mapping.

The max-flow min-cut theorem is applied as follows:

1) For all nodes, create a flow of migration between them. The capacity if the maximum number of neurons could be migrated via them.
2) Since we minimize the extra distance of migration, we use *maximum migrating distance* equal one ($d_{max} = 1; d_{ij} < d_{max}$). *Maximum migrating distance* (d_{max}) is the maximum number of hops that a neuron can migrate. With this $d_{max} = 1$ value, a neuron can only move to one of its four neighbors, which limits the capacity down to the maximum healthy number of neurons of the destination.
3) To allow neurons to migrate more than one hope, we can increase the *maximum migrating distance* value.
4) The movable distance of a neuron could be constrained by the distance to its connected nodes.
5) Once we build all the nodes and the capacity of the flow between nodes.

As shown in Figure 4, we know we can create a specific max-flow min-cut problem by making the flow graph. To solve this problem, we use the Edmonds–Karp algorithm to implement the Ford-Fulkerson method.

The Edmonds-Karp algorithm has the run time complexity of $O(|V||E|^2)$ (E: number of edges, V: number of vertices), which can be translated to $O(N^3)$ for both 2D and 3D network (N: number of nodes). Therefore, the complexity of our mapping is guaranteed as P instead of NP. Meanwhile, heuristic search complexity is $O(N!)$, and ILP is NP-complete. PSO-based approach [27] has the complexity of $O(GKN^2logN)$ (G: number of generation, K: number of particles). Since the number of generations or particles scale with the number of nodes, the PSO approach has a higher complexity than ours (PSO: $O(N^4logN)$ if G and K scales linearly to N; ours: $O(N^3)$). However, the PSO approach [27] requires a massive number of particles which

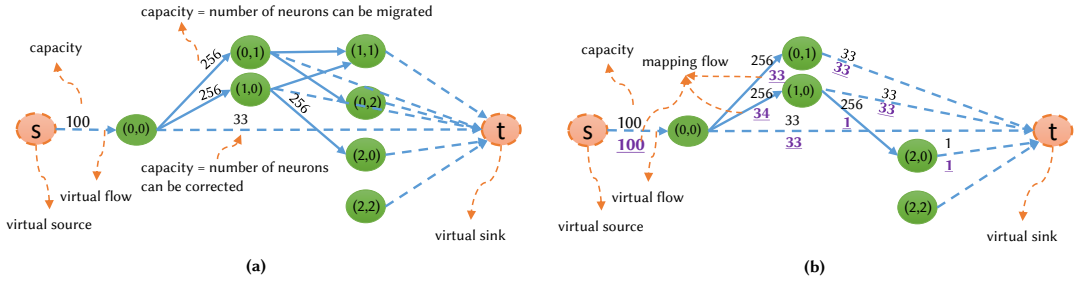


Fig. 4: Flow graph for max-flow min-cut problem: (a) Converted flow from the NoC-based SNN; (b) A solution of max-flow min-cut problem.

makes it has a larger memory footprint than the max-flow min-cut one. The space complexity of the PSO approach [27] and ours is $O(NK)$ and $O(N)$, respectively.

4.2.2 Proposed graph-based algorithm

Algorithm 1 shows our proposed algorithm for tolerating defective neurons. At first, it builds the flow-graph from sources and sinks in lines 2-7. For each node n_i, it adds an edge from the source with the capacity of the number of faulty neurons. As the flow goes out of the source, the algorithm tries to fill the capacity as much as possible. We also connect the node n_i with a virtual sink with the number of spare neurons' capacity. At the end of this part, we built the flow ready for the *node-level recovery*.

In the second part of the algorithm, we first repair the system with *node-level recovery*. Then, we build the flow between nodes by adding the flow from a node to a node within the maximum distance d_{max}. The capacity is the maximum flow between those nodes, which is the minimum value of healthy neurons of the destination. For instance, the node n_j has 120 healthy neurons; the maximum capacity it can gain is 120 since the system can only migrate at most 120 neurons to it.

After completing the flow graph, we perform the Edmonds-Karp algorithm to find the maximum flow and each edge's corresponding flow between the nodes. This number in each edge indicates the number of migrated neurons. The flow of the edge between the nodes and sink is the recovery using spare neurons. After completing the process, we now compare the maximum flow with the number of defective neurons (k). If they are equal, it means the algorithm successfully corrects all k faulty neurons. If they are not equal, it means the max-flow min-cut implementation fails to recover. We will discuss the problem and how we can improve the algorithm in the next section.

Figure 5 illustrates how our algorithm works in a 3D-NoC based neuromorphic system of $N = 3 \times 4 \times 3 = 36$ nodes. Each node consists of 256 neurons, which makes the total number of available neurons $X = 9,216$. Only 9,060 neurons are mapped, which leaves 156 neurons as spares. The system encounters eight faulty nodes with 138 defective neurons cases where only the node (2,2,0) with four faulty and 4 square neurons can complete the recovery using only node-level recovery. By migrating the unmapped neurons to their neighboring nodes, it allows *system-level recovery*. The neighboring nodes now have some unmapped neurons and look for the new nearby nodes. At the end of the algorithm, it successfully maps the 138 faulty neurons, and there are 18 spare neurons left in the system. As shown in Figure 5, the maximum movement of a neuron is only one hop from its original one. For instance, 7 neurons are migrated from (0,0,1) to (0,0,0). The node (0,0,0) also receives four neurons from (1,0,0), which leads to 11 neurons to map. By mapping 11 neurons and having four spares, the node (0,0,0) has seven unmapped and original neurons and migrates them to (0,1,0). By creating chains of migrations within the system, the proposed algorithm helps recover the faulty neuron and minimize the traveling distance of an unmapped neuron original node to the new node.

Algorithm 1: The proposed max-flow min-cut neuron cluster replacement algorithm.

```
   // Build flow graph
1  add source s and sink t;
2  for (node n_i in the system) do
3      add vertex for the node n_i;
4      add edge from the source s to the vertex n_i ;
5      add capacity n_i → s = number of defective neuron in the vertex
         n_i;
6      add edge from the vertex n_i to the sink t;
7      add capacity n_i → t = number of available redundant neurons
         attached to vertex n_i;
8  for (node n_i in the system) do
       // Node-level repair
9      if node n_i has more redundancies than defects then
10         node-level recovery vertex n_i;
11 for (node n_i in the system) do
12     for (node n_j in the system) do
13         if (d_ij ≤ d_max) then
14             add edge from the vertex n_i to the vertex n_j;
15             add capacity n_i → n_j =
                 number of healthy neurons in n_j;
   // System-level repair with Edmonds-Karp
16 while no augmenting path do
17     Breadth first search to find minimum path;
18     Augmenting the found minimum path with capacity;
19     Save the flow;
   // Finish the algorithm and require re-training or
      maintaince if it failed
20 if (max-flow == k ) then
       // done
21     return 0;
22 else
       // The approach fails to correct.
23     return 1;
```

4.2.3 Augmenting Migrating Distance

Although using the max-flow min-cut method can optimize the neuron migration cost, the max-flow min-cut process is

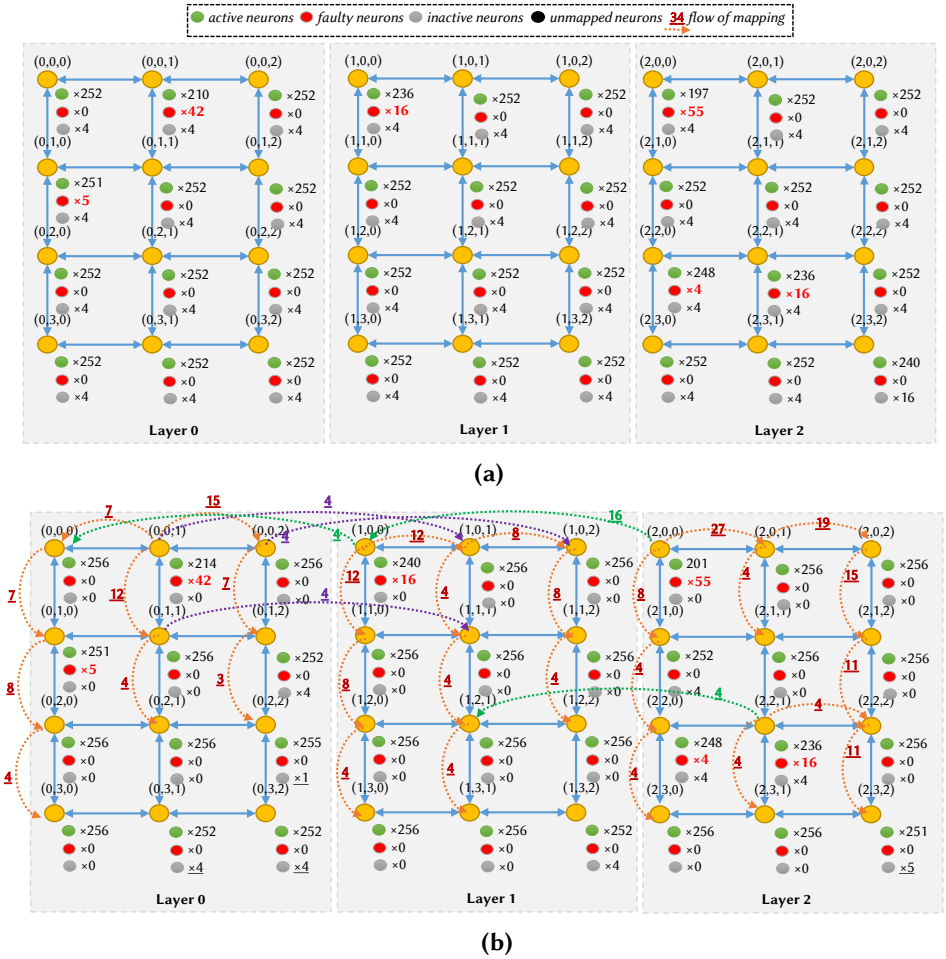


Fig. 5: An illustration of the proposed algorithm: (a) Faulty case; (b) Post-mapping using the proposed algorithm.

not optimal. To increase the minimum cut, we should relax the value of d_{max}; however, it increases the Edmonds-Karp algorithm's complexity (increase number of edges). Based on the max-flow min-cut theorem, the maximum neurons that can be corrected can be limited by the minimal cut of the flow network. Therefore, there is a chance that the system cannot correct as much as neurons as its number of redundancies.

Let us consider the node (0,0) or (0,0,0) in Figure 6. These nodes have 200 faulty neurons and are surrounded by nodes with 210 faulty neurons. After *node-level* recovery, there are 177 unmapped neurons. However, the maximum flow from these nodes to their neighbors is 112 and 168 for 2D and 3D mesh topology, respectively. In this case, the system fails to recover regardless of having redundancies in other nodes. In this fashion, we need to consider a communication cost of two, allowing neurons to move by two hops. The flow graph must be reconstructed for this different d_{max}.

Algorithm 2: Augmenting migrating distance algorithm.

```
d_max = 1;
while (mapping success or d_max > (number of layers + number of rows + number of columns)) do
    run Algorithm 1;
    d_max ++;
```

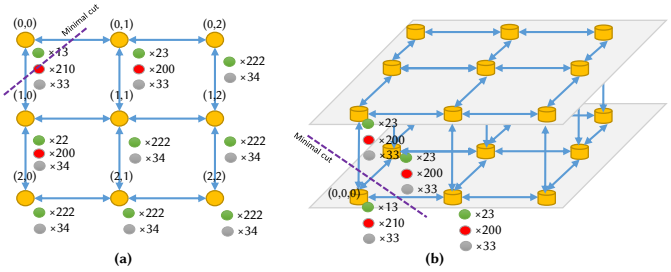


Fig. 6: Examples of the minimal cut drawback: (a) 2D-Mesh; (b) 3D-Mesh

Algorithm 2 shows our augmenting maximum migrating distance algorithm for tackling the problem of the small minimal cut section mentioned above. If the number of faults is larger than the number of spares, the mapping is not successful. Here, we need to run the Algorithm 1 depending on the single or multiple layers SNN. Later, we perform either retraining or maintenance for fitting the SNN to the neuromorphic system.

By gradually increasing the migrating distance, we can find the smallest value to recover the system failure. Therefore, we can balance the trade-off between the maximum migrating distance (d_{max}) and the recovery. Once the d_{max}

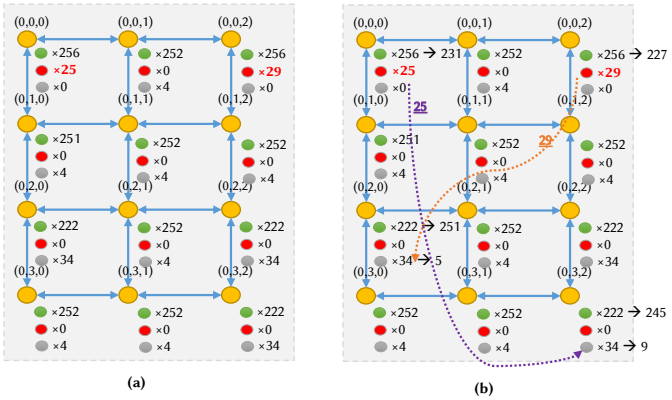


Fig. 7: The *initialize* phase of the Genetic Algorithm: (a) the unmapped and free neurons per node; (b) A randomized mapping solution.

value increases to a maximum distance within the NoC, we can ensure the mapping for the $k \leq R$ cases.

4.3 Proposed Genetic Algorithm

As we presented earlier, a significant issue of the max-flow min-cut is it might fail to map with a maximum migrating distance $d_{max} = 1$. As we increase the d_{max} value, it can lead to mapping for the whole NoC, which can have a considerable size. On the other hand, a metaheuristic such as a Genetic Algorithm can be used to solve the problem in a more holistic manner. Therefore, we also propose a genetic algorithm (GA) for solving the mapping issue in this work. As can be expected, the GA method has higher complexity in time and memory footprint, so it might fail to solve the huge NoC size in a reasonable time. However, the GA approach considers other objectives besides the migration cost (M_{cost}).

Algorithm 3: The proposed Genetic Algorithm for finding cluster replacement.

```
   // initialize phase
1  S1: build the unmapped and free neurons per node;
2  S2: randomize the K mapping solutions;
   // evolve phase
3  for (generation g_i in 1 to G) do
4      S3: calculate cost function for each solution of the population;
5      S4: select the B best out of K solutions based on the cost function;
6      S5: mutate the B best solutions to have new K solutions ;
7      S6: crossover the new K solutions to have new population ;
   // finalize phase
8  S7: calculate cost function for each solution of the population;
9  S8: select the B = 1 best out of K solutions based on the cost function;
```

Algorithm 3 shows the proposed Genetic Algorithm. It consists of three phases: (1) initialize, (2) evolve, and (3) finalize. The initialize phase starts with the first step **S1** where the number of unmapped and free neurons are counted and sent from each node of the system. This one is right after the *node-level recovery* phase. Based on these values, the second step **S2** generates K mapping solutions randomly. This step randomizes a node with free neurons and a node with unmapped neurons from the values in **S1**. At the end of step *S2*, the algorithm generates K legal mapping solutions. They are not optimal solutions and need to be optimized. In the *evolve* phase, the GA method iterates for G generations where each generation repeats four steps. At first, step **S3** computes the cost function for each solution. Here, we can adopt only M_{cost} from Eq. 2. The communication cost F_{cost} is also computed for the selection step $S4$. In $S4$, it ranks the best B solutions out of K, and if they have similar M_{cost} values, their F_{cost} values are considered. Doing so keeps the simplicity of a single objective optimize for GA while still considering migration and communication costs. The formulation of GA is shown in Appendix A.

After getting B best solutions, it goes to two steps: **S5** - crossover and **S6** - mutation. The crossover step **S5** is done by mixing two random mapping solutions. It takes 50% of each parent to generates offspring. By doing so, the offspring can inherit both mappings of its two parents.

There are two types of mutations in the mutation step **S6**. First, it finds an immediate randomized node between two random nodes having a mapping flow. Here, we constrain the immediate node having free neurons is closer to the source node than the destination of the flow. For instance, Fig. 8(a) shows the case where the source node $(0,0,0)$ has 25 unmapped neurons and all are mapped to $(0,3,2)$ - the destination node. Then, it finds the immediate node $(0,2,2)$ with two conditions: (1) there are free neurons the immediate node $(0,2,2)$ and (2) the distance from the source node $(0,0,0)$ to the immediate node $(0,2,2)$ is smaller than the original flow. Here, it remaps the neuron to the immediate node instead of the destination. The result can be seen in Fig. 8(b). The second mutation is to swap the mapping to have a closer migrating distance (smaller M_{cost}). If two flows can have a smaller migrating distance by swapping the destination, the algorithm performs the swap. For instance, Fig. 8(b) shows unmapped neurons in node $(0,0,0)$ are mapped to $(0,2,2)$ and unmapped neurons in node $(0,2,0)$ are mapped to $(0,0,2)$. Here, the migrating distances are four for both flows. However, by switching the destination, we obtain the migrating distances are two for both flows, as shown in Fig. 8(c).

After G generations, the algorithm finalizes by selecting the only best solution (step **S7** and **S8**). This solution is used to perform the mapping method. Since the GA might take a long time to complete, we can also allow early termination of the mapping and use the best-found solution.

In summary, this GA methodology provides an extension for the optimization problem of remapping faulty neurons. While the max-flow min-cut adaptation only focuses on the migration cost, GA allows designers to take other cost functions for the optimization.

4.4 Migration Support Architecture

As discussed in the proposed method for moving neurons, this section illustrates the SNN chip architecture to support these features. In the Network Interface architecture (see Figure 2(c)), there are two paths to map input and output spikes. First, the input spikes are fed to the flit extractor then to the AER decoder to obtain the weight address in the weight memory. To support neuron migration, we also need to support different types of spikes. Second, the migrated neurons also have different IDs than their migrating positions; therefore, remapping the output spikes is necessary.

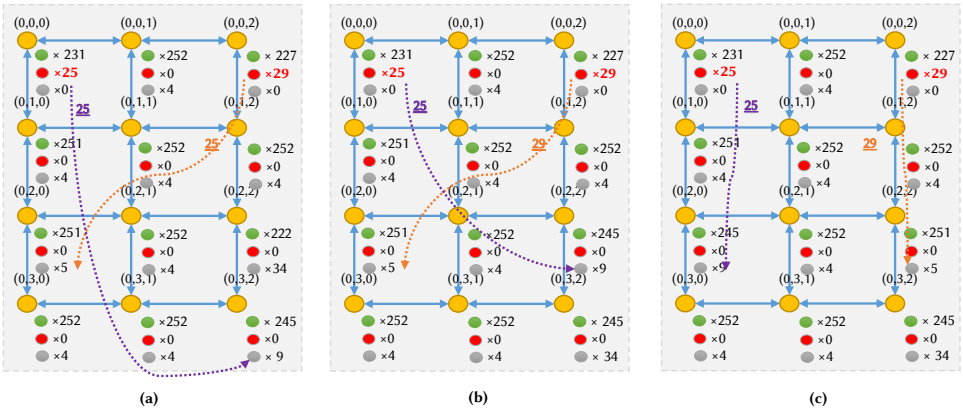


Fig. 8: The *evolve* phase of the Genetic Algorithm: (a) a solution for mutating; (b) mutating by finding a shorter distance for a flows; (c) mutate by swapping destination of a flow.

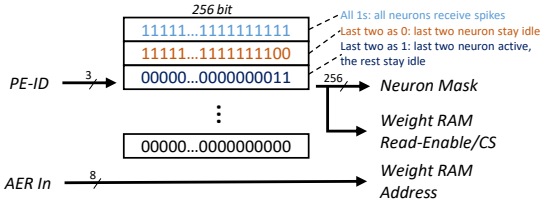


Fig. 9: Input sparsity implementation.

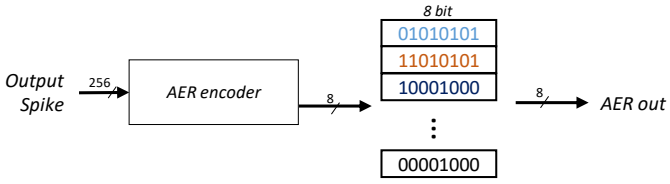


Fig. 10: Output mapping for migrated neurons.

4.4.1 Sparsity Input Spike

Once a neuron migrates to another node, its input spikes could be different from those it immigrated to. In a multi-layer design, migrating a neuron from layer i to layer j will lead to different inputs. Therefore, once an input spike in AER arrives, the Network-Interface must classify into neurons' inputs in different layers/input-types.

We have to note that this is a needed feature with mixed-layer neuron mapping regardless of neuron migration.

In our NoC, we have an additional 3-bit in the spike's flit named PE-ID, which is fed to a programmable Look-up Table to obtain the value of neuron mask (enable signal for weight SRAM). Figure 9 illustrates our method for the input sparsity. The first value of the LUT is all ones; therefore, the spike with PE-ID=0 will cause a weighted spike to all neurons. The second and third elements show the configuration that the last two neurons can receive differently. By using the LUT-based approach, we can make the connections differently for the same SRAM address. The interference weights can be pruned and mapped separately using the same method.

4.4.2 Sparsity Output Spike

Once the faulty neurons are migrated to a different node, their AER value is not similar to or linear with its position. Therefore, we encode the 256-bit output spike vector to an array of 8-bit AER that represents each bit of the spike vector. The output AERs are fed into two Look-up Tables.

Figure 10 illustrates our approach for remapping the output of migrated neurons. By subtracting to the base address, we can obtain the address for AER mapping. The smaller addresses which belong to the original neurons of the node are kept as they are. A multiplexer is used to choose between the mapped AER or the original AER. Assuming we have a spare node without any mapped neuron initially, the programmable LUT must consist of 256 members instead of 8, as shown in Figure 10.

5 Evaluation

This section first provides the evaluation method for our SNNs. Then, it shows the hardware results in 45nm technology. In the following part, we evaluate the performance of *MigSpike* for a multiple layer perceptron (MLP) network. We then also analyze the critical minimal-cut cases and compare the 2D and 3D topology in terms of recovery ability and compare the proposed methods' execution time in *MigSpike* and conventional approaches. The reliability of the system is also evaluated to highlight the benefit of *MigSpike*.

5.1 Evaluation methodology

For the mapping efficiency, we developed the proposed algorithm and performed a comparison with several algorithms. The first algorithm is to remap the SNN with the information of fault. The second one is an A-hop *Greedy Search* (GS), which follows the following rules:

1) Each node corrects itself first. After the self-correction, system-level correction is used.
2) Scan through each node by order of the number of unmapped neurons. The most unmapped neurons node is corrected first.
3) For each correcting node, it looks for closest nodes with a max Manhattan distance of A with spare neurons for recovery.

We choose $A = 1$, and N to show two methods: (1) $A = 1$ only looks for the neighbors for correction; (2) $A = N$ (N: number of nodes) looks for the whole network for recovery. We use *Remap* to denote re-using the mapping method to solve the fault-tolerance issue.

For the hardware implementation, we design the SNN architecture in Verilog HDL. We synthesize and layout with Synopsys Design Compiler and Candence Innovus using the NANGATE 45nm library and the FreePDK3D for the vertical Through-Silicon-Vias. The Through-Silicon-Vias size is $4.06\mu m \times 4.06\mu m$.

5.2 Hardware Complexity

TABLE 1: Hardware complexity of the proposed node.

Module		Area (μm^2)	Power (mW)	Max Freq. (MHz)
Network Interface	AER LUT	16,747	-	-
	Address LUT	20,768	-	-
	Total	72,032	30.4043	699.30
Neuron Cluster		205,608	81.682	751.87
		64KB SRAM	-	-
3D-NoC router [35]		41,739	14.6128	537.63
Vertical TSVs (up and down)		2,901.1136	-	-

Table 1 shows the hardware complexity of the proposed architecture. The proposed NI to support the mapping method is integrated with the neuron cluster and the 3D-NoC router. As shown in Table 1, the additional LUT for AER and Address are 23.35% and 28.83% of the area of the Network Interface. The overhead of these two LUTs is relatively tiny. On the other hand, the NI, which supports migration techniques, only consumes 25.95% of the area cost of the whole tile without the SRAM area (neuron cluster + network interface).

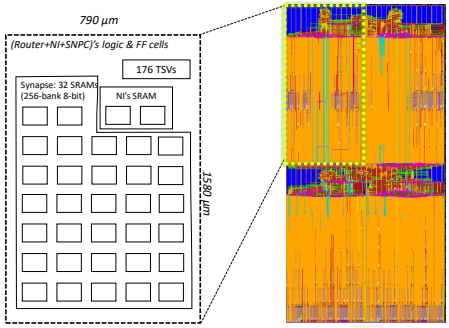


Fig. 11: Layout of a 2x2 NoC-based SNN layer with migration support. A tile's size is $790\mu m \times 1580\mu m$.

Figure 11 illustrates our sample layout for a 2×2 NoC-based SNN layer with migration support. The cluster's configuration is 256 spike input in AER format, 8-bit synapse weight, 32 physical neurons, 32 synapses crossbar for each cluster. Here, each crossbar is implemented with a 256-bank 8-bit dual-port SRAM using OpenRAM. We only integrate 32 neurons per node to have a reasonable Place&Route time and a visual layout. To support 3D-NoC inter-layer interconnect, we use TSV from FreePDK3D45 with the size of $4.06\mu m^2 \times 4.06\mu m^2$ and the Keep-out-Zone is $15\mu m^2 \times 15\mu m^2$ for each TSV. As can be observed in the layer's layout, 80% of the area is for placing macro SRAM. Since the design of the LIF neuron is lightweight, the most complicated part is the crossbar.

On the other hand, the NI requires two dedicated SRAMs for converting the AER from local value to a global one and destination look-up. We can further optimize the design's footprint by reducing the bit-width of a synapse or using an alternative memory approach (eDRAM, STT-RAM, or memristor). Moreover, we add more stacking layers dedicated to memory, which allows us to have a smaller footprint.

5.3 Neuron migration algorithm evaluation

In this section, we evaluate the proposed algorithms (MFMC: *max-flow min-cut* adaption and GA: *Genetic Algorithm*), 1-hop and N-hop, and *Greedy Search* (GS) to understand their efficiency. The *Greedy Search* runs each node once and looks for a spare node within one (1) hop range or in the entire system (N-hop) with the shortest distance. The algorithms are implemented in Java. We insert the faults into the system to evaluate the efficiency of the algorithms. Here, we focus on the communication cost function F_{cost} in Equation 1 and evaluate both 2D and 3D Meshes topology in terms of the migration efficiency. Different system sizes and fault rates are discussed. We evaluate the multi-layer perceptron (MLP) network. The MLP is organized in layers, and the neurons separated by one or more layers are not connected. The input spikes are fed to the router with the smallest indexes (i.e. (0,0) or (0,0,0)). In this evaluation, we measure two major parameters: *(1) mapping rate:* the ability to map the faulty neurons to the spare ones; *(2) average spike transmission cost (F_{cost})*: the average distance of all connections and *(3) Migration cost M_{cost}: the amount of read/write neurons need to adapt the system.*. The configuration of the evaluation is shown in Table 2. Figures 12 and 13 illustrate

TABLE 2: Configuration for the evaluation[1].

Parameter	Value
# neurons per node (E)	256
# nodes (N)	2D-NoC: 4×4 to 16×16
	3D-NoC: $4 \times 4 \times 4$ to $16 \times 16 \times 16$
# spare neurons (R)	0.2×X
# spare node	1
# faults (k)	0.05×X , 0.10×X, 0.15×X, and 0.20×X
SNN # layers	4
SNN configuration[1]	784:0.5*(W-10): 0.5*(W-10): 10

[1] MLP model for MNIST. For example, the SNN configuration for E=256 and 4×4 is 784:1633:1633:10.

the results for the proposed system for 2D-NoC and 3D-NoC configurations (see Table 2). As shown in Figures 12 and 13, our methods can map all faulty neurons to the spare ones regardless of the size or topology. We have to note that the MFMC algorithm is not optimal for communication costs and 1-hop *Greedy Search* can only map around 60% (around 80% with the worst cases) of the faulty neurons. This is because 1-hop *Greedy Search* only runs for once and looks for one mapping solution of its neighbor to fail to map easily. Meanwhile, the N-hop *Greedy Search* and the Genetic Algorithm can map all neurons.

The average F_{cost} (communication cost) also varies between different approaches. Since the 1-hop GS mostly fails

to map the neurons, the average communication distance per neuron is unchanged. For other methods, the average F_{cost} fluctuates between different sizes. However, as we can observe in Figures 12 and 13, they are reduced when we increased the size of the NoC. This due to the fact when we increase the size of the NoC, the impact of moving neurons is reduced. The effects are also more negligible, with smaller fault rates (k values). We can even notice the communication cost maintains with remapping; however, we can observe a slight reduction with the migration-based algorithm. Also, GA seems to have a better average F_{cost} since it reduces that value as the second factor.

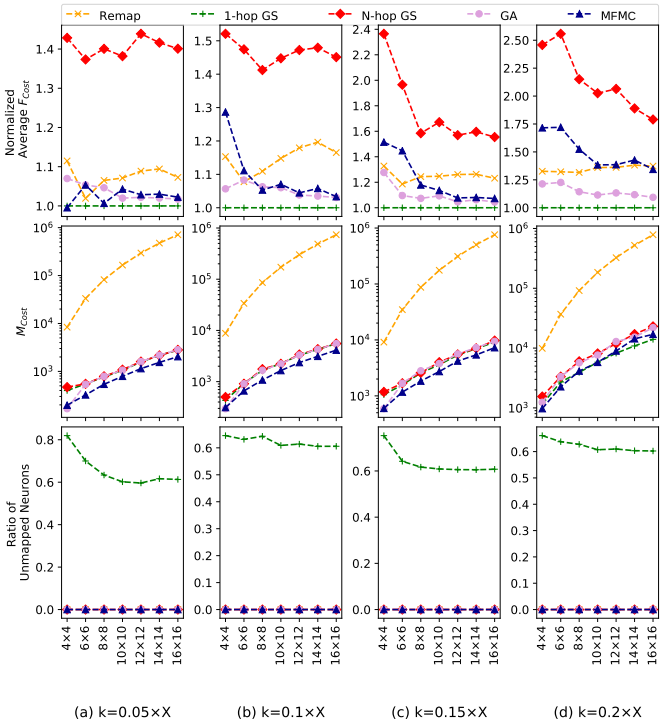


Fig. 12: Output mapping for migrated neurons with random fault patterns in 2D-NoCs. The system has 256 neurons per node; 20% of neurons are spare with 1 redundant node without any allocated neuron at 0% fault rate.

On the other hand, the M_{cost} of MFMC is better than both GA and GS in most cases. However, under [4, 4] and $f = 0.05$ instances, we observe that the M_{cost} of MFMC is worse than the GA. This phenomenon can be explained by the fact that the GA can provide an optimal result (globally or locally) once it converges. Meanwhile, MFMC only tries the maximize the flow between faulty neurons and spare ones. However, once we increase the network's size or change to 3D-NoC, *MigSpike* easily dominates GA and GS. While GS is not an optimal approach, GA might need adjustments to find the optimal solution (i.e., different evolving methods or more generations). However, as we will discuss in the execution time evaluation, GA costs a long time to execute, limiting its efficiency.

5.4 Critical minimal-cut cases

As we presented in Figure 6, one of the significant drawbacks of the max-flow min-cut method is the case where

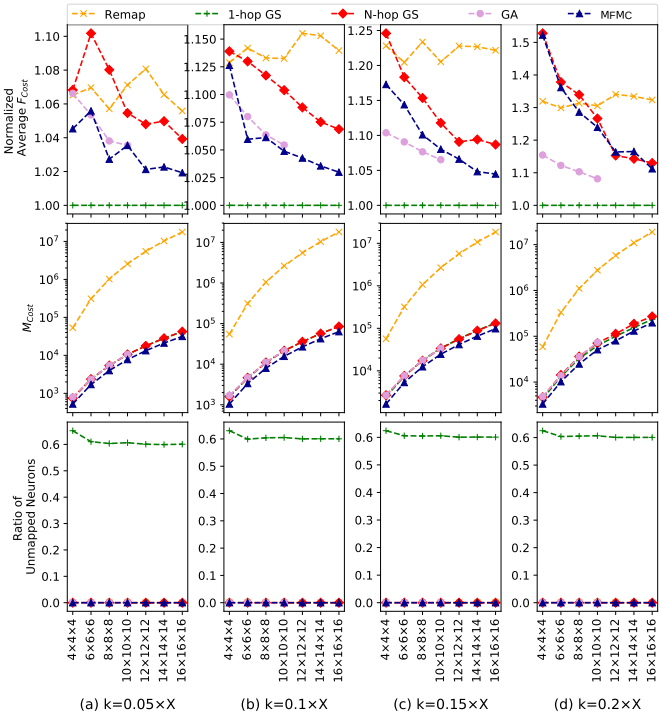


Fig. 13: Output mapping for migrated neurons with random fault patterns in 3D-NoCs. The system has 256 neurons per node; 20% of neurons are spare with 1 redundant node without any allocated neuron at 0% fault rate.

the minimal-cut is too tiny and creates the bottle-neck. The groups' border can be recovered with MFMC; however, the central node cannot make it. With typical Ford-Fulkerson implementation, we can see that around 20% of the faulty node cannot be remapped, as shown as MFMC in Figure 6. By relaxing the value of d_{max}, the MFMC-AMD system can map 100% of the faulty node.

We also evaluated the system with 2D and 3D-mesh topology configurations. We set the number of nodes per system like 16, 32, 64, 128, and 256. As illustrated in Fig. 16, the 3D configurations provide better communication and migration costs. Even with fault, the communication cost of the 3D configurations is still lower than the 2D configurations. This is due to the fact the 3D topology has a small number of hops between nodes. Moreover, since it has more path diversity, the migration cost of 3D-NoC is also better. For instance, with 16, 32, 64, 128 and 256 nodes and 20% fault rates, the migration costs of 3D-NoCs are only 81.64%, 75.05%, 74.32%, 71.11% and 72.24% of 2D-NoC ones.

5.5 Execution Time

Table 3 shows the execution time for different network sizes. The evaluation was performed on Intel Xeon E5-2620v4, 16 GB, and Windows 10. The algorithms are written in Java without thread parallelism. Table 2 shows the complete configuration of the evaluation. The Genetic Algorithm is the slowest method since it requires a considerable amount of population and generations. The GA implementation requires 200 generations and 100 parents for complete

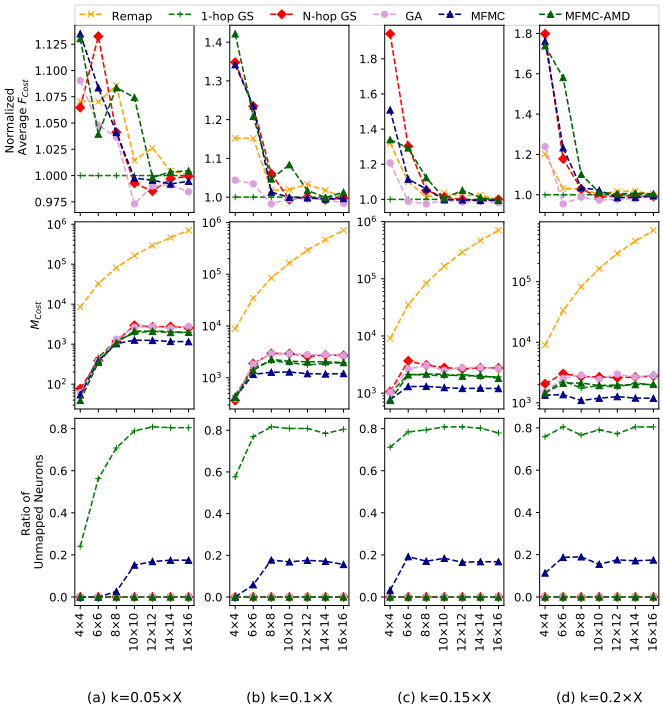


Fig. 14: Output mapping for migrated neurons with the minimal-cut cases in 2D NoCs. The system has 256 neurons per node; 20% of neurons are spare with 1 redundant node without any allocated neuron at 0% fault rate. MigSpike-AMD: the augmenting migrating distance d_{max} method.

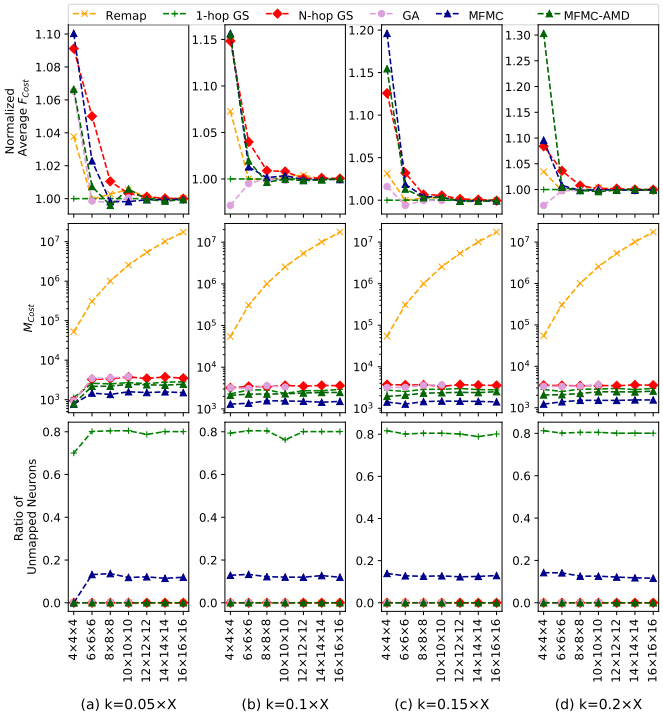


Fig. 15: Output mapping for migrated neurons with the minimal-cut cases in 3D NoCs. The system has 256 neurons per node; 20% of neurons are spare with 1 redundant node without any allocated neuron at 0% fault rate. MigSpike-AMD: the augmenting migrating distance d_{max} method.

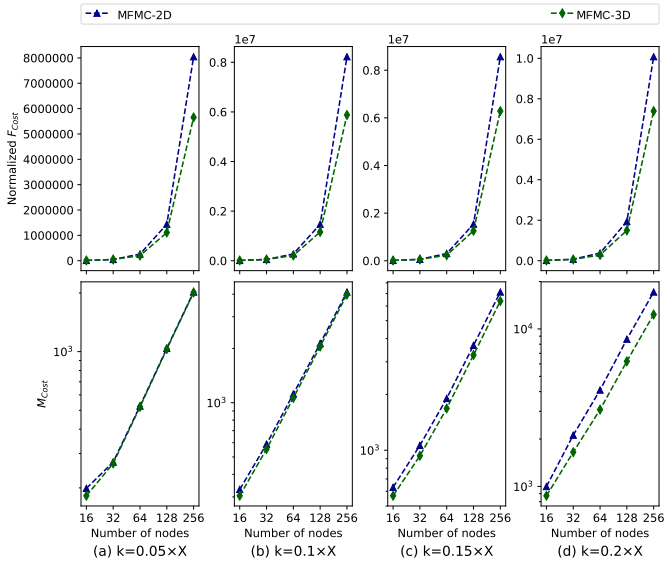


Fig. 16: Comparison between 2D and 3D NoCs.

mapping, making it slower than other approaches and not scalable in terms of execution time.

In comparison to the two *Greedy Search* implementations, our methods are slower. This is due the fact that *1-hop GS* and *N-hop GS* have the time complexity of $O(N)$ and $O(N^2)$, respectively. Meanwhile, our approaches provide better migration costs (M_{cost}) than the greedy approach. This leads to a lower repair time in our algorithms. We also perform the same benchmark with W=1000 and E=128 to compare with Load Balancing [32] and the greedy approach using Kernighan-Lin graph partition [38]. The source code is obtained from [32], which targets optimization of the transmission between neurons instead of migration time. As we perform in the same machine as mentioned above, the remapping, 1-hop GS, N-hop GS, GA, and *MigSpike* take 14.3 μs, 44.6 μs, 65.400 μs, 3.856 s, and 310.1 μs, respectively, to complete the migration solution.

5.6 Reliability

Figure 17 illustrates the Mean Time to Failure (MTTF) of the MigSpike and the non-protected system under the 2D and 3D Network-on-Chip systems, respectively. In [39], the MTTF of both non-protected and redundancy-based systems can be modeled using the Markov-state model. We assume that the fault rate (λ_{neuron}) of a neuron is a constant value. The MTTF of a neuron is:

$$\mathbf{MTTF}_{neuron} = \frac{1}{\lambda_{neuron}} \tag{3}$$

The MTTF of a non-protected system of X neurons are:

$$\mathbf{MTTF}_{non-protected} = \frac{1}{X\lambda_{neuron}} \tag{4}$$

The MTTF for a k-fault tolerance system [40] is:

$$\mathbf{MTTF}_{FT} = \frac{1}{\lambda_{neuron}} \Sigma_{i=W}^{W+k} \frac{1}{i} \tag{5}$$

TABLE 3: Execution time of migrations algorithm for 2D and 3D NoCs[1].

	Size	Remap	1-hop GS	N-hop GS	GA[2]	MFMC
2D NoCs	[4,4]	20.1 μs	90.3 μs	113 μs	5.916 s	351.4 μs
	[6,6]	38.2 μs	378.2 μs	421.3 μs	53.826 s	672.9 μs
	[8,8]	74.3 μs	0.640 ms	1.155 ms	140.392 s	1.202 ms
	[10,10]	216.5 μs	1.526 ms	2.076 ms	327.070 s	2.640 ms
	[12,12]	369 μs	3.413 ms	3.499 ms	640.914 s	5.032 ms
	[14,14]	608 μs	5.438 ms	6.150 ms	1220.911 s	7.428 ms
	[16,16]	932.5 μs	7.698 ms	8.713 ms	1932.054 s	11.097 ms
3D NoCs	[4,4,4]	39.2 μs	780.100 μs	942.8 μs	141.509 s	1.276 ms
	[6,6,6]	116.3 μs	7.349 ms	8.387 ms	1498.355 s	8.771ms
	[8,8,8]	394.5 μs	13.487 ms	14.076 ms	3.178 h	24.604 ms
	[10,10,10]	1.200 ms	32.151 ms	44.706 ms	15.237 h	63.545 ms
	[12,12,12]	3.365 ms	101.105 ms	106.223 ms	N/A [3]	116.658 ms
	[14,14,14]	8.333 ms	184.655 ms	204.551 ms	N/A [3]	208.800 ms
	[16,16,16]	19.938 ms	236.766 ms	291.288 ms	N/A [3]	442.986 ms

[0] The execution only takes into account the computation time for finding the new mapping. Other calculations such as setting up or configuration generation are not counted.
[1] System configuration: E=256, W = 0.8*X, k = 0.2*X.
[2] Genetic Algorithm configuration: 100 parents, 20 bests, 40 crossover and 40 mutations per generation, 200 generations.
[3] GA in 3D-NoC with the size from [12,12,12] is infeasible to perform due to extremely long execution time and large allocated memory.
[4] Linear mapping: the neurons with lower indexes are mapped to the nodes with lower indexes.

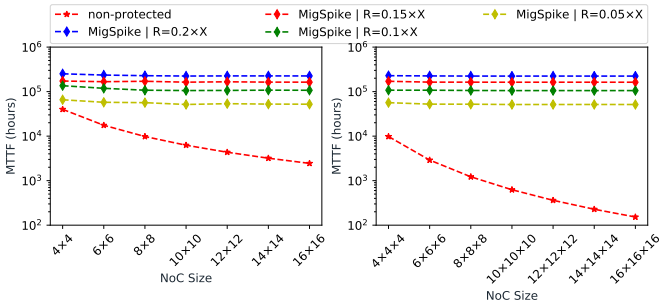


Fig. 17: Mean Time to Failure of the proposed system under 2D and 3D NoCs.

Where W is the number of needed neurons for the application, and k is the maximum number of correctable neurons. Note that $0 < k \leq R$ with R is the number of redundant neurons. We set λ_{neuron} as 1000 per 10^9 hours (1 per 114.08 years), which is later scaled with the proper technology, design, and operating condition. As shown in Figure 17, the *MigSpike* design can maintain the MTTF in different network sizes. With the spare rate of 0.2, the MTTF values of *MigSpike* are larger than 2E5 hours (22.82 years). Meanwhile, the non-protected systems fail under 2E4 hours (2.28 years) with $N \geq 64$.

5.7 Discussion

In the previous evaluations, we have presented the efficiency of *MigSpike*. Despite the obtained advantages, some challenges should be addressed to enhance further the recovery ability of *MigSpike*, as discussed hereafter.

If number of faulty neurons k is larger than the number of redundancies R, the proposed algorithm is unable to correct with either *system-level* or *node-level* recovery. To deal with this situation, there are two possible options: (a) *retraining* the system with a reduced number of neurons (b) *migrating* the whole system to a new chip (i.e., [25] varied the threshold voltage to adapt with defective weight; or Dropout and Dropconnect [41] can make smaller neurons/connections network). If the system supports multi-chips, we have another option to *plug-and-migrate* a new cluster of neurons and migrate the SNN task to this new neuron. Once the new neurons are attached, they are attached to flow, and a migration process is performed by the proposed method. In either case, further investigation needs to ensure the proper approach and the execution time to retrain and reconfigure the system can be overwhelming; therefore, we do not consider them in this work.

We can observe that the Algorithm 1 does not take into account the layer index of the neurons in a multi-layers SNN into its consideration. The value of Equation 1 in layer-less models of SNN or Liquid State Machine seems not to be affected by moving the neuron to its neighbors. To optimize the distance of neurons among layers of SNN, we can consider each layer as a sub-system and perform the Algorithm 1 for each layer. After completing for each layer of the SNN, we can move on to the rest of the network by only performing for the system's unmapped neurons. The GA approach can consider the communication cost; therefore, it can maintain the layer shape if it reduces the communication cost.

We can undoubtedly adopt the conventional approach for protecting the SNN system. For instance, adding ECC to detect and correct faulty SRAM cells and using Triple Modular Redundancy (TMR) to ensure the LIF neurons' correctness are two viable options. However, both methods are not cost-efficient as TMR triples the area cost, and ECC makes a significant overhead because SRAM cells take the major footprint of the chip.

As the proposed architecture and algorithm based on migrating the neurons, this work might be adapted to other domains (non-spiking NN or distributed computing). However, two key factors allow us to tailor the method for SNN. First, neuron or computing unit of SNN is a meager cost; therefore, adding redundant computing units is not critical as in other computing domains. Second, SNN transmits spikes in AER format where only the global address of the firing neuron is sent. This allows high-flexible mapping as the local firing neuron address is converted into a global one using a LUT, as in Fig. 10.

We have not evaluated the convolution SNN in the mapping solution in the evaluation section as our hardware architecture does not support convolution and pooling layers. Despite the lack of support for hardware spiking CNNs, the mapping method is still suitable for being applied to CNNs as the problem formulation can be used. The details on tolerating faults in spiking CNNs will be investigated in the future work of MigSpike.

Despite the limitations mentioned above, we believe that *MigSpike* still provides a low migration cost solution for tolerating faults in NoC-based SNN systems. The exhibited overhead in a 3D-NoC implementation is also reasonable, which makes *MigSpike* an up-and-coming solution for integration into highly reliable 3D ICs.

6 Conclusion

This work presents a design and implementation for a fault-tolerant Spiking Neural Network by adding spare

neurons and an augmented algorithm for recovery. As we demonstrated in the evaluation, the proposed algorithm can help recover up to 100% spare neurons regardless of fault distribution. At the same time, it requires reasonable area overhead on Network Interface and execution time for finding the proper mapping. In terms of migration, *MigSpike* decreases the cost by 10.19× and 96.13× times compared to the remapping approach. We also augmented the max-flow min-cut approach to tackle its fundamental drawback when multiple nearby nodes' faults occur. Further study in learning algorithms, advanced memory technology such as RRAM or STT-RAM could help improve the SNN design. Multiple chips system is another approach for future works. Moreover, adapting the algorithm into spiking CNN and a better dataset is another future work.

APPENDIX A
GENETIC ALGORITHM FORMULATION

The input/state for our genetic algorithm is:

$$\text{State} = \{S_{(i,j,k)}\} \tag{6}$$

where $S_{(i,j,k)}$ is the free/faulty state of the node indexed ((i,j,k)) in the NoC. If $S_{(i,j,k)} > 0$, the node ((i,j,k)) has $S_{(i,j,k)}$ free neurons. If $S_{(i,j,k)} < 0$, the node ((i,j,k)) has $|S_{(i,j,k)}|$ unmapped neurons. For instance, Fig. 7 (a) with a 3×4 NoC has the input (or initial state) of $3 \times 4 = 12$ elements as follows:

$$\text{State}_{init} = \{-25, 4, -29, 4, 4, 4, 34, 4, 34, 4, 4, 34\} \tag{7}$$

The goal of GA is to convert that state to a final sate State_{final} with all non-negative elements (no unmapped neurons). A encoding of GA is as follow:

$$\text{Solution} = \{M_{(i,j,k)\rightarrow(m,n,l)}\} \tag{8}$$

where $M_{(i,j,k)\rightarrow(m,n,l)}$ is the number of neurons migrated from node (i,j,k) to node (m,n,l) in the NoC. For instance, Fig. 7 (b) shows a solution for Fig. 7 (a) with the following values:

$$\text{Solution}_{\text{Fig.7(a)}} = \begin{bmatrix} 0,0,0,0,0,0,0,0,0,0,0,\mathbf{25} \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,\mathbf{29},0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,\mathbf{-29},0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ 0,0,0,0,0,0,0,0,0,0,0,0 \\ \mathbf{-25},0,0,0,0,0,0,0,0,0,0,0 \end{bmatrix} \tag{9}$$

As can be seen in Eq. 9, the solution is an array of 12×12 for representing the mapping between 12 nodes. There are two positive values and two negative values representing the mapping of Fig. 7 (b). The rest of the solutions are zeros as no more mapping existed. Here, we can calculate the migration cost by considering the distance between nodes and the positive value.

ACKNOWLEDGMENTS

This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.01-2018.312.

REFERENCES

[1] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[2] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, January 2018.

[3] F. Akopyan *et al.*, "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, Oct 2015.

[4] S. B. Furber *et al.*, "The SpiNNaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, May 2014.

[5] B. V. Benjamin *et al.*, "Neurogrid: A mixed-analog-digital multi-chip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, May 2014.

[6] J. Schemmel *et al.*, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 1947–1950.

[7] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," *eLife*, vol. 8, p. e47314, Aug. 2019.

[8] H. Hazan *et al.*, "BindsNET: A machine learning-oriented spiking neural networks library in Python," *Frontiers in Neuroinformatics*, vol. 12, p. 89, 2018.

[9] M. Ogbodo, T. Vu, K. Dang, and A. Ben Abdallah, "Light-weight spiking neuron processing core for large-scale 3D-NoC based spiking neural network processing systems," in *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2020, pp. 133–139.

[10] T. H. Vu, O. M. Ikechukwu, and A. Ben Abdallah, "Fault-tolerant spike routing algorithm and architecture for three dimensional NoC-based neuromorphic systems," *IEEE Access*, vol. 7, pp. 90 436–90 452, 2019.

[11] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in computational neuroscience*, vol. 9, p. 99, 2015.

[12] N. Rathi, G. Srinivasan, P. Panda, and K. Roy, "Enabling deep spiking neural networks with hybrid conversion and spike timing dependent backpropagation," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=B1xSperKvH

[13] T. L. Michalka, R. C. Varshney, and J. D. Meindl, "A discussion of yield modeling with defect clustering, circuit repair, and circuit redundancy," *IEEE Transactions on Semiconductor Manufacturing*, vol. 3, no. 3, pp. 116–127, 1990.

[14] P. V. Bhanu *et al.*, "Fault-tolerant network-on-chip design with flexible spare core placement," *J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 1, Jan. 2019.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[16] M.-Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Res. Dev.*, vol. 14, no. 4, pp. 395–401, 1970.

[17] M. Hsiao, D. Bossen, and R. Chien, "Orthogonal latin square codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 390–394, 1970.

[18] A. Neale, M. Jonkman, and M. Sachdev, "Adjacent-MBU-tolerant SEC-DED-TAEC-yAED codes for embedded SRAMs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 4, pp. 387–391, 2014.

[19] Ilyoung Kim *et al.*, "Built in self repair for embedded high density SRAM," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, 1998, pp. 1112–1119.

[20] B. W. Denkinger *et al.*, "Impact of memory voltage scaling on accuracy and resilience of deep learning based edge devices," *IEEE Design & Test*, 2019.

[21] A. Prodromou *et al.*, "Nocalert: An on-line and real-time fault detection mechanism for network-on-chip architectures," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 60–71.

[22] A. Ben Ahmed and A. Ben Abdallah, "Adaptive fault-tolerant architecture and routing algorithm for reliable many-core 3D-NoC systems," *Journal of Parallel and Distributed Computing*, vol. 93-94, pp. 30 – 43, 2016.

[23] K. Constantinides *et al.*, "BulletProof: A defect-tolerant CMP switch architecture," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* IEEE, 2006, pp. 5–16.

[24] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design & Test*, vol. 36, no. 5, pp. 44–53, 2019.

[25] A. P. Johnson *et al.*, "Homeostatic fault tolerance in spiking neural networks: a dynamic hardware perspective," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 2, pp. 687–699, 2017.

[26] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM journal of research and development*, vol. 6, no. 2, pp. 200–209, 1962.

[27] P. K. Sahu *et al.*, "Application mapping onto mesh-based network-on-chip using discrete particle swarm optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 2, pp. 300–312, 2013.

[28] A. Namazi *et al.*, "A majority-based reliability-aware task mapping in high-performance homogenous NoC architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 1, pp. 1–31, 2017.

[29] G. Kim *et al.*, "Optimal distribution of spiking neurons over multicore neuromorphic processors," *IEEE Access*, vol. 8, pp. 69 426–69 437, 2020.

[30] S. Li *et al.*, "SNEAP: A fast and efficient toolchain for mapping large-scale spiking neural network onto NoC-based neuromorphic platform," *arXiv preprint arXiv:2004.01639*, 2020.

[31] X. Jin, "Parallel simulation of neural networks on spinnaker universal neuromorphic hardware," Ph.D. dissertation, The University of Manchester (United Kingdom), 2010.

[32] A. Balaji *et al.*, "Mapping spiking neural networks to neuromorphic hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 76–86, 2019.

[33] Y. Ji, Y. Zhang, H. Liu, and W. Zheng, "Optimized mapping spiking neural networks onto network-on-chip," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2016, pp. 38–52.

[34] K. N. Dang and A. Ben Abdallah, "An efficient software-hardware design framework for spiking neural network systems," in *The International Conference on Internet of Things, Embedded Systems and Communications (IINTEC 2019)*, 2019.

[35] K. N. Dang *et al.*, "Scalable design methodology and online algorithm for TSV-cluster defects recovery in highly reliable 3D-NoC systems," *IEEE Transactions on Emerging Topics in Computing*, in press.

[36] T. H. Vu, Y. Okuyama, and A. Ben Abdallah, "Comprehensive analytic performance assessment and K-means based multicast routing algorithm and architecture for 3D-NoC of spiking neurons," *J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 4, pp. 34:1–34:28, Oct. 2019.

[37] C. Frenkel *et al.*, "A 0.086-mm^2 12.7-pJ/SOP 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS," *IEEE transactions on biomedical circuits and systems*, vol. 13, no. 1, pp. 145–158, 2018.

[38] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[39] M. L. Shooman, *Reliability of computer systems and networks: fault tolerance, analysis, and design*. John Wiley & Sons, 2003.

[40] K. N. Dang *et al.*, "Reliability assessment and quantitative evaluation of soft-error resilient 3D network-on-chip systems," in *2016 IEEE 25th Asian Test Symposium (ATS)*. IEEE, 2016, pp. 161–166.

[41] Y. Sakai, B. U. Pedroni, S. Joshi, S. Tanabe, A. Akinin, and G. Cauwenberghs, "Dropout and DropConnect for reliable neuromorphic inference under communication constraints in network connectivity," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 658–667, 2019.

Khanh N. Dang is currently an assistant professor at Vietnam National University Hanoi (VNU), Hanoi, Vietnam. He received his B.Sc., M.Sc., and Ph.D. degree from Vietnam National University Hanoi (VNU), University of Paris-XI, and The University of Aizu, Japan, in 2011, 2014, and 2017, respectively. Dr. Khanh N. Dang was a visiting researcher at the University of Aizu in 2019 and 2020-2021. He has served as a TPC co-chair of MCSoC 2019/2021 and APCCAS 2020. His research interests include System-on-Chips/Network-on-Chips, 3D-ICs, neuromorphic computing, and fault-tolerant systems.

Nguyen Anh Vu Doan is currently working as a Postdoctoral Researcher with the Chair of Integrated Systems, Technical University of Munich, Munich, Germany. He received the Ph.D. degree in engineering from the Université Libre de Bruxelles, Bruxelles, Belgium, in 2015. His research interests include design space exploration, multi-objective optimization, and multicriteria decision aiding.

Abderazek Ben Abdallah is a Professor in the graduate and undergraduate schools of computer science and engineering at the University of Aizu, Aizu-Wakamatsu, Japan. He is concurrently Head of the Division of Computer Engineering at the school of computer science and engineering. He has a Ph.D. degree in computer engineering from the University of Electro-Communications in 2002. Dr. Ben Abdallah conducts research in the area of computer systems and parallel computing, with an emphasis on adaptive/self-organizing systems, on/off-chip interconnection networks, and reliable multicore systems-on-chip. His current research interest lies in studying neural processing systems with a particular focus on spike-based neural network dynamics and spike-based learning. He is a senior member of ACM and IEEE.