A Defect-Tolerant Accelerator for Emerging High-Performance Applications

Olivier Temam

INRIA Saclay, France
olivier.temam@inria.fr

Abstract

Due to the evolution of technology constraints, especially energy constraints which may lead to heterogeneous multi-cores, and the increasing number of defects, the design of defect-tolerant accelerators for heterogeneous multi-cores may become a major micro-architecture research issue.

Most custom circuits are highly defect sensitive, a single transistor can wreck such circuits. On the contrary, artificial neural networks (ANNs) are inherently error tolerant algorithms. And the emergence of high-performance applications implementing recognition and mining tasks, for which competitive ANN-based algorithms exist, drastically expands the potential application scope of a hardware ANN accelerator.

However, while the error tolerance of ANN algorithms is well documented, there are few in-depth attempts at demonstrating that an actual hardware ANN would be tolerant to faulty transistors. Most fault models are abstract and cannot demonstrate that the error tolerance of ANN algorithms can be translated into the defect tolerance of hardware ANN accelerators.

In this article, we introduce a hardware ANN geared towards defect tolerance and energy efficiency, by spatially expanding the ANN. In order to precisely assess the defect tolerance capability of this hardware ANN, we introduce defects at the level of transistors, and then assess the impact of such defects on the hardware ANN functional behavior. We empirically show that the conceptual error tolerance of neural networks does translate into the defect tolerance of hardware neural networks, paving the way for their introduction in heterogeneous multi-cores as intrinsically defect-tolerant and energy-efficient accelerators.

I. Introduction

Even though transistor count keeps increasing, in the past few years, ever more stringent technology constraints are forcing to deeply revisit micro-architectures. After energy constraints motivated a shift from high clock frequency processors to homogeneous multi-cores, the lack of voltage scaling is breeding the so-called "Dark Silicon" [43], [18] constraint where only a fraction of transistors can be used simultaneously due to the limited on-chip power budget. That constraint, in turn, is likely to induce a novel shift towards heterogeneous multi-cores, composed of a mix of cores and accelerators, where only a few accelerators are used at any given time. As a result, increasing attention should be devoted to the design of accelerators. However, beyond energy, the growing number of *defects* is becoming another major constraint that will just as severely affect the design of micro-architectures [6], [1]. Within the context of heterogeneous multi-cores, the challenge is then to find not only energy-efficient, but also *defect-tolerant* accelerators.

One of the key challenges of the design of accelerators is to find the right balance between application scope and efficiency: entirely reconfigurable circuits are known to lack energy efficiency [31], while the scope of ASICs is naturally too limited. We need to design accelerators which can cover a reasonable share, though not all, of the application spectrum; the combination of several such accelerators, rather than a single accelerator, shall bring sufficient application coverage. The GreenDroid project [49], for instance, argues for the introduction of a large number of custom circuits in order to tackle a broad range of tasks at a fraction of the energy budget of a core. Another approach is to consider a multi-purpose accelerator, capable of accelerating the key routines of many applications, such as the loop accelerator of Clark et al. [13]. While highly energy efficient, not only these designs are not meant to be defect tolerant, but they are even more susceptible to defects than homogeneous multi-cores, where core redundancy at least provides a form of defect tolerance. In this article, we introduce and synthesize a multi-purpose accelerator which can tolerate multiple defects, which can potentially implement the computational kernels of some of the emerging high-performance tasks using alternative ANN-based algorithms, and, like custom circuits, which can achieve more than two orders of magnitude better energy efficiency than general-purpose cores.

What are the emerging high-performance applications ? Recently, Intel [17] has attracted the attention of the community to RMS (Recognition, Mining and Synthesis) applications, as some of the most important emerging high-performance applications. That effort partly motivated the

development of the PARSEC benchmark suite [5] by Princeton University. Many of the PARSEC benchmarks rely on four categories of algorithms: *classification*, *clustering*, *statistical optimization* and *approximation*. While the PARSEC benchmarks rely on a varied set of techniques, for each of these four kinds of algorithms, ANNs have been shown to provide a competitive alternative [23]. For at least half of the tasks implemented in PARSEC benchmarks (especially Recognition and Mining tasks), we can envision replacing the core computational task with an alternative ANN-based algorithm, so that the application scope of an ANN accelerator is potentially broad.

Why focusing on Artificial Neural Networks ? The key benefit of ANNs is that they are intrinsically tolerant to transient or permanent errors. If this property can be translated into the hardware design itself, then a hardware ANN can become a defect tolerant accelerator. While the potential defect tolerance of ANNs is certainly not a new notion, there are few in-depth attempts at evaluating the defect tolerance capability of ANNs down to the transistor level. Some of the most recent attempts at evaluating the defect tolerance capability of hardware ANNs [20] still rely on abstract models, e.g., stuck-at synapses. In this article, we assess the impact of transistor-level defects on the functionality of ANNs. We show that the impact of such defects can be significantly different from stuck-at synapses. We then propose a novel hardware ANN design, which is inherently defect tolerant because it does not *spatially fold* synapses into an SRAM bank, and neurons into a few hardware operators, as was commonplace in past [25] or recent [33] designs. On the contrary, we show that *spatially expanded* designs bring significant defect tolerance benefits; they also improve energy efficiency by bringing storage (synapses) closer to computational operators (neurons). We then precisely assess the defect tolerance capability of our hardware ANN design, i.e., the impact of defects on the accelerator accuracy. We evaluate the accuracy using a set of classification tasks from the UCI machine-learning repository [3], but the ANN design would be the same for approximation, or clustering tasks; some modifications might be necessary for optimization tasks (because they might rely on the Hodgkin-Huxley neuron model [23]), but we leave such extensions for future work.

While hardware ANNs are an old concept [25], [40], they fell out of favor in the 1990s for three main reasons: (1) from a mathematical standpoint, ANNs were outperformed by other algorithms such as Support Vector Machines (SVMs), (2) at a time when scientific computing formed the bulk of high-performance applications, the application scope of ANNs was restricted, (3) hardware ANNs could not keep pace with the the speed of software versions run on processors with rapidly progressing clock frequency, exactly like many of the parallel architectures with custom processors at the same period. However, all three situations have drastically changed: (1) Deep Networks [34], i.e., ANNs made of a large number of wide layers, have recently been shown to outperform SVMs [34], (2) a significant fraction of the recognition and mining tasks can potentially be implemented using ANNs as their core algorithms, so an ANN-based accelerator might be able to tackle a large share of emerging high-performance workloads, (3) the processor clock frequency has mostly stalled, so that a hardware accelerator will now retain an advantage of about two orders of magnitude in energy and performance over a software model run on a processor.

As a testimony to the growing adequation between neural network-based accelerators and important high-performance applications, several industrial hardware neural network designs have recently appeared. For instance Chakradhar et al. from NEC [10] propose to implement a convolutional hardware neural network using a modified FPGA, which is a fast, albeit moderately energy efficient, design for tackling a broad range of recognition tasks. IBM has even more recently introduced the Cognitive Chip [47], which is meant to be an energy efficient design, but it is apparently not geared towards defect tolerance.

II. A Spatially Expanded Hardware ANN Accelerator

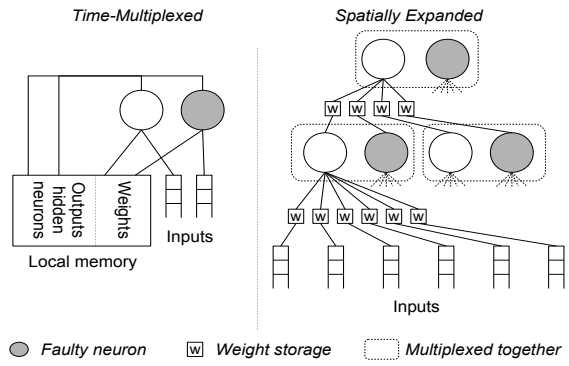


Fig. 1: *Time-Multiplexed vs. spatially expanded network.*

Artificial Neural Networks. We first briefly recall the main principles of ANNs. The most traditional form of ANNs are Multi-Layer Perceptrons (MLPs); we consider a 2-layer MLP with one hidden layer, plus the input layer (it contains no neuron). MLPs are feed-forward networks, where information flows from the input layer ($l = 0$) to the output layer ($l = 2$). Each neuron performs the following computations. Let y_j^l the output of neuron j at layer l, $y_j^l = f\left(o_j^l\right)$ where $o_j^l = \sum_{i=0}^{N_{l-1}} w_{ji}^l y_i^{l-1}$, w_{ji} is the synaptic weight between neuron i in layer $l-1$ and neuron j in layer l, N_l is the number of neurons in layer l, and f is the activation function, often the sigmoid $f(x) = \frac{1}{1+exp^{-x}}$.

We use back-propagation [23], the most popular training algorithm. Formally, the weights are updated as follows: $w_{ji}^l(t+1) = w_{ji}^l(t) + \eta\delta_j^l(t)y_i^{l-1}(t)$, where t is the training iteration, η is the learning rate, and δ_j^l is the error gradient, i.e., the direction of the error. At the output layer, the gradient

expression is $\delta_j^l(t) = f'(o_j^l(t)) \times e_j^l(t)$, where e_j^l is the error (the difference between the network output and the expected output), and in the hidden layer, the gradient expression is $\delta_j^l(t) = f'(o_j^l(t)) \times \sum_{k=0}^{N_{l+1}} \delta_k^{l+1}(t) w_{kj}(t)$, where f' is the derivative of f.

Spatial expansion vs. time-multiplexing. Many previous and recent ANN designs for high-performance are time-multiplexed [25], [33]: only a few hardware neurons are implemented, and synapses are stored in a separate memory bank, see Figure 1. At each time step, a few neurons of a layer are temporarily mapped to the network by fetching the corresponding synapses; the intermediate output is stored to serve as input for next layer neurons; executing a whole neural network requires several time steps.

In a spatially expanded feed-forward ANN, the ANN structure and operation are similar to that of a conceptual ANN: the synapses are stored in individual distributed storage locations at the level of each neuron, and all neurons are mapped into hardware; the data flows combinationally from the input layer to the output layer, see Figure 1. Besides the resulting short synaptic weights access latency, the internal synaptic weights bandwidth is high, not constrained by a single synaptic storage structure. Finally, decentralized synaptic storage means the synapses (data) are located close to the neurons (operators), resulting in significant energy benefits [21].

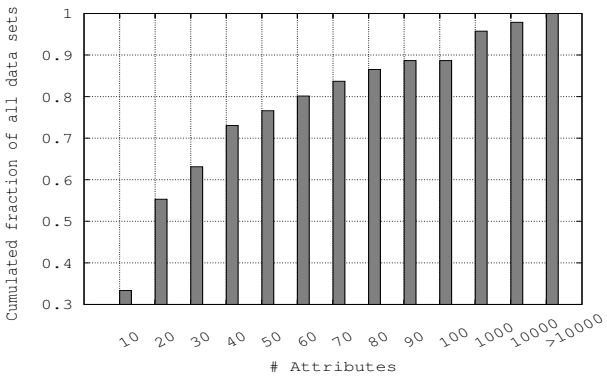


Fig. 2: *Distribution of UCI data sets as a function of the number of attributes.*

Is it possible to spatially expand ANNs for useful tasks ? A key parameter that will influence the network size is the number of *attributes* of the network, i.e., the number of inputs. We have collected the number of attributes of all 135 applications of the UCI machine-learning repository [3]. The example cases in that repository are contributed by researchers and engineers from various domains in order to stimulate machine-learning research on their applications.

The distribution of the number of inputs for all 135 UCI applications in Figure 2 shows that more than 92% of UCI data have less than 100 attributes. A neural network with 90 inputs can capture most of these cases (our design choice).

A few applications in the UCI repository have much more attributes, e.g., more than 10000, and machine-learning researchers are often using input sets with a large number of attributes, such as the MNIST [35] database (handwritten digits recognition, 784 attributes) in order to exercise their machine-learning algorithms. However, our goal is not to exercise machine-learning techniques but to use data corresponding to actual problems, from a broad variety of domains to which machine-learning techniques are applied, hence the choice of UCI.

Still, the proposed spatially expanded ANN accelerator can cope with a larger number of attributes. In that case, we simply consider our network as a sub-network and partially time-multiplex the larger network over it; however, the size of our (sub-)network is such that time-multiplexing is considerably reduced compared to moderately expanded designs.

Defect tolerance and scalability. In a fully time-multiplexed network with an SRAM bank for storing synapses, a significant share of the logic is dedicated to the time-multiplexing process itself: address decoder, routing synapses to operators, results back to storage, etc. A faulty transistor within this control logic would wreck the accelerator.

A spatially expanded neural network has no control logic except at the input/output. Because the synaptic weights storage is distributed at the level of each neuron, there is no central storage, and no decoding logic for read operations (we still require write decoding logic in our implementation). As a result, one or several faulty neurons or synapses usually have no noticeable effect after re-training.

This defect-tolerance property also improves scalability. Because control logic is vulnerable, it should be implemented with larger transistors as the technology node scales down; in a spatially expanded network, the smaller fraction of control logic has a lesser impact on scalability.

If the spatially expanded network is used in a partially time-multiplexed mode, it remains tolerant to defects. However, a defect at a given hardware neuron would affect all the neurons of the application network mapped to it, effectively multiplying the number of defects by as much as the multiplexing factor.

III. Injecting Transistor-Level Defects

A. Hardware Faults

The classic hardware fault model is the *stuck-at* model [32], where an input gate signal is considered stuck at 0 or 1. This fault model is appropriate for test purposes because the goal is to find faults, and then to *suppress* them. So the goal is not so much to accurately capture the gate behavior for any input, but to do so with enough inputs that the fault can be detected. Even though we leverage test techniques, our goal is different: we will not suppress the faults, we will use the circuit *with* the faults. However the actual behavior of a faulty ANN circuit can potentially influence the ability of the ANN to cope with faults, and the ability of the training algorithm

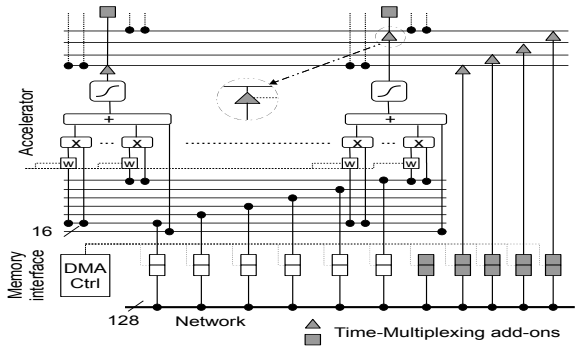


Fig. 3: *Accelerator implementation (memory interface & hidden layer).*

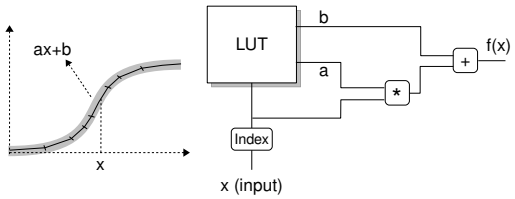


Fig. 4: *Sigmoid: piecewise approximation and hardware implementation.*

to silence out faulty parts. Therefore, unlike for test purposes, we need to model as accurately as possible the behavior of the faulty circuit, not just determine that it is faulty; and we will highlight that logic gate-level hardware faults can exhibit a significantly different behavior than transistor-level hardware faults. Failing to accurately account for the behavior of faulty hardware ANNs means that claims of their defect tolerance would remain fairly baseless, or at least unproven.

Traditionally used in circuit-level testing, the stuck-at gate model has been adopted by the micro-architecture community in the form of *latch bits* stuck at 0 or 1 [41], [37]. Such a model accurately describes faults occurring at state elements (latches, registers, SRAM), but it inaccurately describes faults occurring within logic elements (control and logic operators in processors or ASICs). However, in a spatially expanded neural network, logic operators account for a large share of the accelerator footprint. Recently, a more accurate gate-level model has been proposed by Li et al. [36], where hardware faults are modeled as stuck (or delayed) inputs of *logic gates*. While this model is a significant progress over latch-level bit flips, the actual behavior of logic blocks resulting from *transistor-level* defects can often be more complex than stuck-at and delayed inputs of logic gates, in part because they occur *within* the complex gates often used in modern CMOS design [42].

The two main types of transistor-level defects are *shorts* and *opens* (either full or partial ones, between the source and the drain, or between the gate and the drain/source) [32], [4]. For instance, shorts can correspond to insufficient metal being removed during photolithography, and opens to an excess of metal being removed (or an excess of insulating material). Shorts and opens can force transistor paths to be (or appear) *stuck open or closed*, or create *bridges* between transistors, or induce delays. In many cases, the impact of these hardware faults on the logic gate function cannot be modeled using a stuck logic gate input: for instance, the logic gate function will be changed, or it will be transformed into a state element, or it can depend on free floating devices, etc.

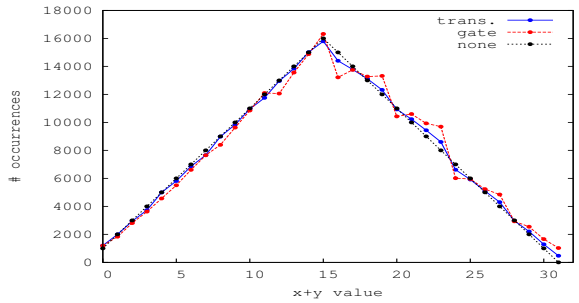
As a result, we factor in the impact of transistor-level defects at the level of logic gates. Starting from an RTL gate-level logic function, we transform it into a transistor schematic, inject any number of transistor-level shorts or opens, and reconstruct the new transformed logic function (or state element) corresponding to the altered schematic.

In order to illustrate the behavior discrepancy between gate-level defect injection and transistor-level defect injection, we inject a random number of defects in adders and multipliers (the key logic operators of the neural network); we use 4-bit adders and multipliers in this example, while the network uses 16-bit operators. The defects are randomly spread over the operator bits, and within each 1-bit operation, over all transistors. Then we consider all possible combinations of the two 4-bit inputs, and collect the distribution of the values of the output. We present the 4-bit input pairs in random order in order to avoid any special behavior related to the memory property induced by some faults. The results are shown in Figure 5 for 1, 5 and 20 defects injected in the 4-bit adder, and 20 defects injected in the 4-bit multiplier. The dotted (black) line corresponds to the distribution of values of the error-free sum (see `none`), the dashed (red) line to the defects injected at the gate level, and the solid (blue) line to the defects injected at the transistor level. We repeat this experiment 1000 times (injecting different defects each time), and collect the total distribution over these 1000 iterations.

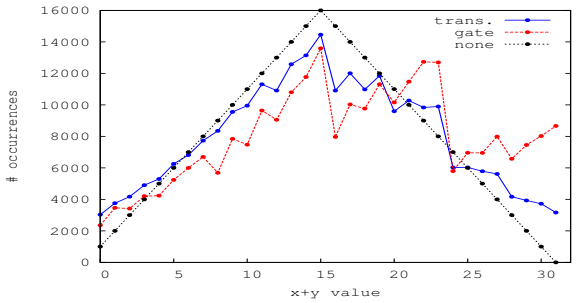
For 1 defect, the behavior of the 4-bit adder is barely affected. As the number of defects increases from 5 to 20, the distribution of values becomes increasingly different from the actual sum distribution. However, the profile for transistor-level defects remains closer to the error-free profile than the gate-level defects profile. At 20 defects, the gate-level profile is barely related to the error-free profile. A similar behavior can be observed for the multiplier. In other words, for such fundamental operators as the adder and the multiplier, it can be observed that transistor-level and gate-level defects induce significantly different behavior.

B. From Transistor Defects to Logic Expression

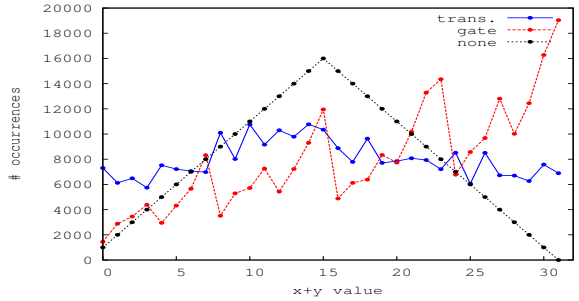
The goal of this section is twofold: to illustrate why transistor-level and intra-gate defects can induce significantly different gate-level behavior than stuck-at gate input defects, and to explain how a logic function (or state element) can be reconstructed after transistor-level defect injection, in order to emulate the behavior of a faulty circuit.



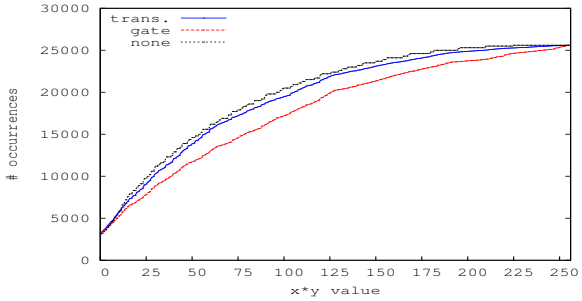
4-bit adder, 1 defect



4-bit adder, 5 defects



4-bit adder, 20 defects



4-bit multiplier, no defect, 20 gate-level defects, 20 transistor-level defects

Fig. 5: *4-bit adder and 4-bit multiplier.*

Consider the CMOS representation in Figure 7 of the logic gate $\overline{(a+b).(c+d)}$ of Figure 6. In a defect-free gate, the logic expressions Z_P and Z_N formed by the transistors of respectively the N and P channel networks are $Z_P = Z$ and $Z_N = \overline{Z}$, since the P channel network connects V_{dd} to Z and the N channel network connects V_{ss} to Z.

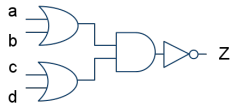


Fig. 6: *Logic circuit* $\overline{(a+b).(c+d)}$.

Let us first consider some of the differences between the gate-level and transistor-level fault model. A hardware fault in the gate-level model could, for instance, correspond to input a of one of the OR gates to be stuck at 1. At the transistor level, an open at the drain of transistor 1, and a short between the source and drain of transistor 7 (the two transistors where a is the gate input, see Figure 7), for instance, could induce the same behavior.

A more likely situation is that only one of these two defects occurs. Such a defect is going to break the symmetry between the N and P CMOS channel networks, and change the gate logic function. If, for instance, the open at the drain of transistor 1 occurs, the path through transistor 1 is stuck off, and Z can only be connected to V_{dd} through transistors 2 and 4 when $c = 0$ and $d = 0$; so the P region of the gate

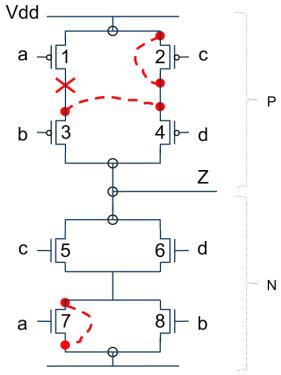


Fig. 7: *Transistor schematic with shorts (dashed lines) and opens (crosses).*

now behaves like logic gate $Z_P = \overline{c}.\overline{d}$ instead of the original $Z_P = \overline{(a+b).(c+d)}$. The N part of the gate retains its original expression $Z_N = \overline{\overline{(a+b).(c+d)}} = (a+b).(c+d)$. So, while in a defect-free gate, Z_P and Z_N can never have the same value ($Z_P = \overline{Z_N}$), in the presence of defects, it can happen that either the N and P parts of the gate are both conducting, or conversely, both disconnected from respectively V_{ss} and V_{dd}. For instance, for $a = b = 0, c = d = 1$, $Z_P = Z_N = 0$.

Shorts can result in similar or even more complex cases. For instance, the short between the source and drain of transistor 2 is equivalent to the transistor being stuck on, so

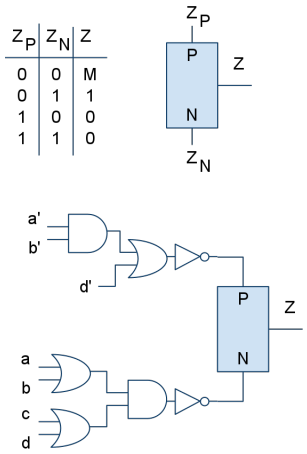


Fig. 8: *B-Block and its truth table; logic expression of Z using a B-Block for the example of Figures 6,7 with short between source and drain of transistor 2.*

Z can be connected either when $a = b = 0$ or when $d = 0$, so $Z_P = \overline{a}.\overline{b} + \overline{d} = \overline{(a+b).d}$. The short connecting the drain of transistor 1 and the drain of transistor 2 would affect the logic expression of Z_P in a more subtle way. Transistors 1 and 2 are now in parallel, as well as transistors 3 and 4, so Z is connected to V_{dd} if $a = c = 0$ or $a = b = 0$ or $c = b = 0$ or $c = d = 0$, so the expression of Z_P is changed to $Z_P = \overline{a}.\overline{d} + \overline{a}.\overline{b} + \overline{c}.\overline{b} + \overline{c}.\overline{d} = \overline{(a.c) + (b.d)}$. Again, there are cases where $Z_N = Z_P$.

What happens when $Z_N = Z_P$ and how to reconstruct a logic function expression with asymmetric N and P channel networks ? When $Z_P = 1$ and $Z_N = 1$ simultaneously, both V_{ss} and V_{dd} are connected to Z but the path from ground (Z_N) dominates and the output is pulled to the logic value 0 [27].

When $Z_P = 0$ and $Z_N = 0$ simultaneously, there is no path from either V_{ss} or V_{dd} to Z, so the output retains its previous value, there is a *memory* effect [32].

As a result, with asymmetric N and P channel networks, not only the logic expression of Z has changed (because of case $Z_N = Z_P = 1$) but it has also become a state element (because of case $Z_N = Z_P = 0$). This state element can be modeled with a *B-block* gate [27] and its truth table is shown in Figure 8. Using this B-block, it is possible to give a logic representation of the asymmetric transistor schematic resulting from opens and shorts, as shown in Figure 8. The main feature is that the B-block allows to separately reconstruct the N and P-channel network logic functions, and to later combine them so as to obtain the gate logic value.

A delay due to partial shorts and opens, or gate to drain/source shorts, etc, can also be inserted anywhere in the N or P channel networks, breaking up the corresponding logic function into two sub-functions. It takes the form of a state element that stores the line value and propagates it at the next transition(s).

The process for injecting transistor-level defects and obtaining an altered logic expression can be summarized as follows:

1) Logic to schematic. The first step is to convert the logic function into a CMOS transistor schematic, and to inject defects, see Figure 9(a).
2) TLogic nodes. During the reconstruction process, from the transistor schematic to the logic function, the notion of TLogic (Transistor-Logic) nodes is introduced in order to represent gates with a source and drain, i.e., partially converted transistors; these hybrid gates correspond to transistor bundles, on which transistor-to-logic conversion rules can still be applied as if they were transistors. See a TAND in Figure 9(c).
3) Splitting connections. Hardware defects introduce asymmetries in the schematic that can prevent the reconstruction into a logic function. In order to overcome the asymmetry, some connections need to be split, sometimes at the cost of virtually replicating transistors. See connection j_2 split into j_2, j'_2 in Figure 9(d).
4) Bypasses. Some hardware defects, such as bridges, can result in bypasses which void the functional role of some transistors. As a result, these transistors can be eliminated when reconstructing the transistor schematic into a logic function. See bypass $j_1 \rightarrow j_4$ resulting in the elimination of serial transistors e and b, in Figure 9(e).
5) TLogic to Logic. The final step consists in converting remaining TLogic gates into normal logic gates, essentially removing the source and drain connections. See Figure 9(f,g).

We have implemented these different steps as one automated process in order to assess different neural networks organizations and operators (e.g., different sigmoid functions, different implementations of arithmetic operators, etc).

Currently, one of the main limitations of the approach is that transistor layout is not factored in. In the future, we plan to introduce a simplified form of layout in order to know which transistors are physically close, and thus most susceptible to endure bridging effects, for instance.

IV. Hardware ANN Implementation

In this section, we briefly introduce the hardware neural network in which we will be injecting defects.

Overview. A scaled down version of the spatially expanded hardware neural network accelerator is shown in Figure 3; the actual network contains 90 inputs, 10 hidden neurons and 10 outputs. The number of inputs and outputs are selected based on the typical number of attributes and classes in the examples of the UCI repository, as explained in Section II. The number of hidden neurons (10) is the best trade-off between accuracy and cost for the example cases we consider, see Section V. The network is fully connected.

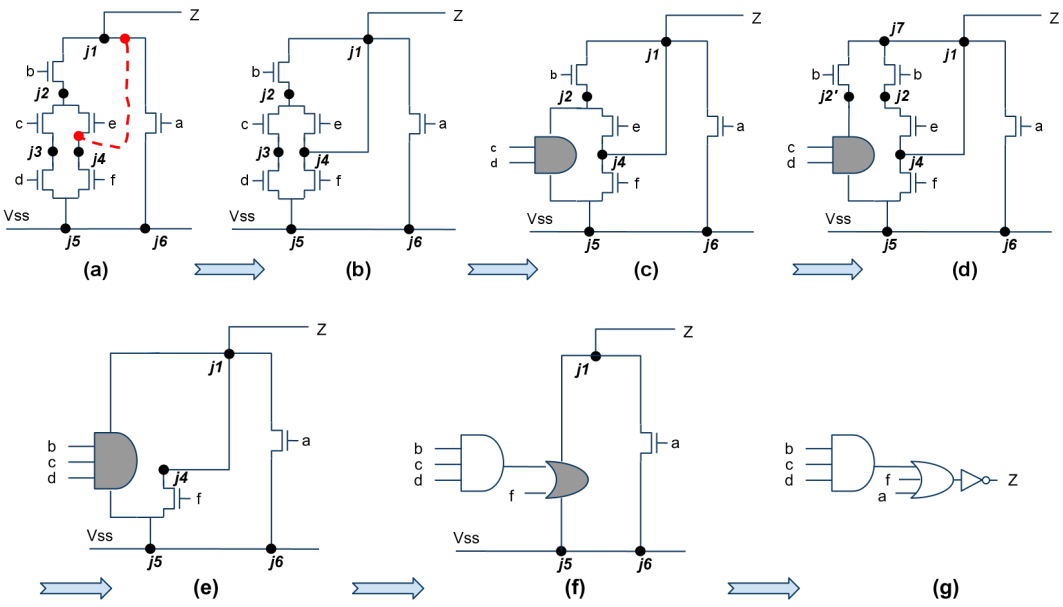


Fig. 9: *(a) Bridge defect between connections j_1 and j_4; (b) resulting transistor schematic; (c) first replacement of transistors* c *and* d *with TAND(c,d); (d) splitting connection j_2; (e) bypass $j_1 \rightarrow j_4$ voids transistors* b *(between j_7 and j_2) and* e *(between j_2 and j_4), and transistor* b *is serially connected with TAND(c,d); (f) transforming Tlogic into plain logic; (g) reconstructed logic function.*

There are two main types of application scenarios for classifiers: off-line training where the classifier is trained then delivered (and periodically retrained when necessary), and on-line training where the classifier is continuously trained. The latter scenario usually corresponds to smart sensors and industrial control, while the former scenario corresponds to the high-performance processing of large volumes of data and is the most typical scenario. We focus our accelerator design on that scenario, and perform the ANN training in a companion core, though using the forward hardware logic. Therefore, if a defect occurs within the hardware ANN, the retraining will factor in the faulty elements. The accelerator can also be extended to include training hardware for tackling both the on-line and off-line scenarios.

Input/Output.

Fetching input rows. We use a DMA for efficiently fetching the input rows. The DMA uses a 2-latch buffer for each input in order to let the accelerator use one row while it fetches the next. Similarly, the DMA uses a 2-latch output buffer for each output. The accelerator and the DMA communicate through a simple 2-signal handshaking protocol (ready/accept).

Writing synaptic weights during training. We use the same interface for fetching input rows, and for writing synaptic weights (during training). For that purpose, each internal wire is connected to both the input of multipliers and the synaptic weights input latches, see Figure 3. Each neuron of layer l is reloaded one by one: all its N_{l-1} synaptic weights are loaded, then stored. A write signal w_j^l, for neuron j at layer l, is activated by the DMA. Then the synaptic weights of the next neuron are loaded, etc.

Fixed-Point computations. Even though many software implementations of ANNs rely on floating-point computations, this is usually overkill [16]. Fixed-Point computations with as little as 8 bits have been shown to achieve similar accuracy for a broad range of problems [24], and as a result, many past hardware implementations have relied on fixed-point arithmetic with a limited number of bits. In order to consider a sufficiently broad set of problems, we opt for a 16-bit design, with a 6-bit integral part, 10-bit decimal part. We empirically checked that, for all the problems we consider in this article, this 16-bit design allows to achieve the same accuracy as a floating-point design.

Activation function. The sigmoid function implements the non-linear behavior of artificial neurons. We implement this function using a piecewise linear approximation using a small look-up table ($x \rightarrow f(x) = a_i \times x + b_i$), where each table entry contains the pair of coefficients (a_i, b_i), see Figure 4. Each segment i applies to a limited range of the input value x. We empirically observed that approximating the function with 16 segments has no noticeable impact on the network accuracy compared to the original sigmoid. The activation function characteristics are reported in Table III using the Synopsys Design Compiler at 90nm; the latency corresponds to the computation of $f(x)$ (read access to the LU, multiplication then addition).

Partial time-multiplexing. For the problems which do not fit in the spatially expanded network, we can still resort

to time-multiplexing. All neurons of the network are then considered to belong to one large layer. As a result, we must now be able to directly feed inputs to neurons in the output layer, or collect the output of neurons in the hidden layer. For that purpose, we alter the design in several ways, see `add-ons` in grey in Figure 3. We add as many input latches as the number of hidden neurons, and connect these input latches to the internal buses between the hidden and output layers. These input latches can feed input values directly to the neurons of the output layer. And the output of the hidden neurons is connected to the additional output latches. As a result, all neurons can now be used as if they were part of a single layer.

Processing one input row of a neural network which contains N times the number of neurons of the hardware neural network requires at least N times the delay necessary to process one input row in the hardware neural network.

V. Experimental Framework

We implement the accelerator and the memory interface in Verilog and synthesize them with the Synopsys Design Compiler, using the TSMC 90nm library.

We have developed two other software models of the artificial neural network. One model is meant for network training and testing, training is performed using back-propagation; this model has been validated against the FANN library [44]. The model is also used to inject hardware defects as later explained in Section VI-C. The second model is just meant for feed-forward testing within the processor simulator. It is oriented towards very fast execution, and it is a trimmed down version of a neural network which performs only the same operations as the hardware version, albeit in C.

For the superscalar processor, we use Wattch/SimpleScalar and configured the processor so that it roughly emulates an Intel Stealey (A110) core [14]. Our goal is to compare the accelerator against a low-power high-performance core, such as the Intel Atom, using the same 90nm technology node. While the Atom was first implemented in the 45nm process, the Stealey (A100 and A110), derived from the Pentium M and implemented at 90nm, is considered a predecessor of the Atom. The Stealey is a 800MHz, 3W processor.

While the initial version of Wattch was designed for a 350nm technology [8], the 100nm technology has been added to version 1.02. We use the fudge factor to further slightly scale down the technology to 90nm. Wattch has an approximate evaluation of leakage power, however at 90nm, leakage power is still roughly less than 30% of total power [38]; moreover, we will see later on that the energy difference between the processor and the accelerator is such that this lack of precision on leakage power is tolerable. It could be more significant with much smaller technology nodes.

The trimmed down software model is compiled into Alpha binary using the `crosstools` [28], and the `-O` optimization level. The overhead of the initialization and readout steps are rendered negligible by executing the ANN computations for 1000 input rows.

Hyper Parameter	*From*	*To*	*Step*
# Hidden neurons	2	16	+2
# Epochs	100	3200	*2
Learning rate	0.1	0.9	+0.1
Momentum	0.1	0.9	+0.1

TABLE I: *Hyper-Parameter space.*

Problem	*Description*	*# In, # Out*	*η, # Epochs, # Hidden*
`breast`	Breast cancer diagnostic	30, 2	0.1, 200, 14
`glass`	Glass oxides identification (forensic)	9, 6	0.1, 800, 10
`ionosphere`	Radar returns from ionosphere	34, 2	0.3, 100, 6
`iris`	Plants classification	4, 3	0.2, 100, 8
`optdigits`	Handwritten digits recognition	64, 10	0.1, 200, 14
`robot`	Failure detection	90, 5	0.2, 1600, 6
`sonar`	Metal vs. Rock sonar returns	60, 2	0.1, 100, 10
`spam`	Email spam identification	57, 2	0.1, 800, 6
`vehicle`	Vehicle silhouettes recognition	18, 4	0.1, 400, 6
`wine`	Wine origin based on chemicals	13, 3	0.2, 1600, 4

TABLE II: *# Inputs (attributes) and outputs (classes), and best hyper-parameters (η = learning rate,* hidden = *optimal # of hidden neurons).*

We have selected 10 applications from the UCI Machine-Learning repository to serve as a "benchmark" suite [3]. The two selection criteria were application diversity and having less than 100 attributes. The data characteristics are summarized in Table II. For each application, we have searched the optimal accuracy within a hyper-parameter space indicated in Table I. We report the main optimum training hyper-parameters (epoch, learning rate) in Table II. All accuracy measurements provided in the remainder of the article uses 10-fold cross validation. The network is trained using MSE (Mean Squared Error) though we report accuracy measurements.

VI. Evaluation

A. Accelerator area, energy, latency and accuracy characteristics

We evaluate the area cost, latency and energy of the network using the Synopsys Design Compiler. The energy includes both the dynamic and leakage power. The neural network accelerator characteristics are provided in Table III. At 90nm, the accelerator footprint is 9.02mm2, the total dissipated power is 4.70W, the time to process one input row is 14.92ns, and the total energy spent per input row is 70.16nJ per row.

Characteristic	*Accelerator*	*Activation*	*Interface*
Time (in ns)	14.92 ns	2.84 ns	
Freq. (in Mhz)			800
Area (in mm^2)	9.02	0.017	0.047
Power (in W)	4.70	0.0019	0.0054
Energy per input row (in nJ)	70.16	0.0053	0.0021

TABLE III: *Accelerator, memory interface and activation function characteristics.*

Characteristic	*Value*
Clock frequency	800MHz
# cycles per row	19680
Avg. power per cycle (in W)	2.78
Energy per row (in nJ)	68388

TABLE IV: *Processor execution characteristics.*

Memory interface and key logic. We have separately evaluated the characteristics of the memory interface because it is part of the key logic, i.e., logic that cannot be scaled down like the ANN logic because it must be defect-free. The memory interface includes the DMA controller, while the write signals for synaptic weights latches, the control signals for connecting the output of the hidden neurons, and the additional input latches (for partial time-multiplexing) to the internal wires are included in the key logic.

The target network configuration has 90 attributes, so $90 \times 16 = 1440$ bits must be loaded to feed one input row to the neural network. To maximize the accelerator throughput, 1440 bits must be fetched every 14.92 nanoseconds, i.e., 11.23GB/s; these numbers are in line with the bandwidth of the recent Intel Quick Path Interconnect [39], for instance (12.8GB/s in one direction). We assume two links capable of bringing 64 bits each every cycle, and must clock the interface at 754MHz, or higher, for achieving the maximal processing bandwidth of the accelerator (11.23GB/s); we use 800MHz. The memory interface characteristics are reported in Table III.

At 90nm, the interface and key logic area accounts for a negligible amount of the overall accelerator area. Assuming a factor of area size reduction of 2 across technology nodes, with the interface and key logic area remaining constant, it would only account for less than 10% of the overall accelerator area after 4 technology generations (22nm), and 25% at the 6th generation (11nm). Moreover, the number of neurons in the accelerator can also be scaled up across generations, further reducing the relative overhead.

B. Accelerator vs. processor

Characteristics of processor execution. As mentioned in Section V, we simulate a trimmed down software model so that it performs almost the same operations as the hardware accelerator.

Since we assume an identical underlying memory system, and since we do not aim at comparing DMA vs. memory hierarchy and hardware prefetching, we restrict the comparison to the accelerator and the processor. For that reason, we assume a perfect 1-cycle L1 cache, and subtract the power and energy used by the L1 and L2 caches. The 1-cycle perfect cache aims at bringing the processor-to-memory latency at the level of a DMA, and to avoid biasing the comparison towards the neural network with a more efficient memory subsystem.

We clock the processor at 800MHz, the maximal frequency of the Intel Stealey, and the same as for the DMA.

The performance characteristics of the processor run are shown in Table IV for a 90-10-10 software ANN. The number of cycles and energy are provided per input row, i.e., for each set of 90 inputs. The measured accelerator power (4.70W) is actually higher than the processor power (2.78W) due to the large number of computing elements in the accelerator. But the energy consumed by the accelerator (70.16 nJ per row) is drastically lower than that of the processor (68388 nJ per row) because processing one input row requires only 14.92ns in the accelerator, thanks to the massive parallel multiplications/additions and circuit-level parallelism, while it requires 19680 cycles in the processor, i.e., 24600ns at 800MHz.

While high, this energy ratio is consistent with previous research, such as the work by Hameed et al. [21], which reports a slightly lower energy ratio of 500x for an H264 decoder run on a general-purpose core and implemented as a custom circuit; Chung et al. [12] report lower ratios of about 100x, but both studies factor in the impact of memory accesses, while the numbers reported in Table III are for the accelerator only, as mentioned above. Other accelerators can also provide significantly better energy efficiency than cores, especially GPUs and FPGAs. While precise energy ratios can vary wildly across applications, architectures and technology, GPUs and FPGAs can roughly be an order of magnitude more energy efficient than CPUs [12], and FPGAs are at least an order of magnitude less energy efficient than ASICs [12], [31].

C. Defect Tolerance

In order to evaluate the impact of defects, we randomly pick one of the logic operators or latches within the input and hidden layers, and one 1-bit operator or wire within the target operator or latch. The operator is then altered because of the defect, and replaced by a reconstructed logic gate function, according to the process described in Section III-B. In our high-level ANN software model used for training and testing, it is possible to mark a neuron as having one or several defect(s) for a specific operator, in which case a software function is called to perform that operator in place of the native operator.

The impact of such defects, as a function of the number of defects, is shown in Figure 10. For each number of defects, we report the accuracy obtained after re-training, in order to demonstrate the network capacity to silence out defects.

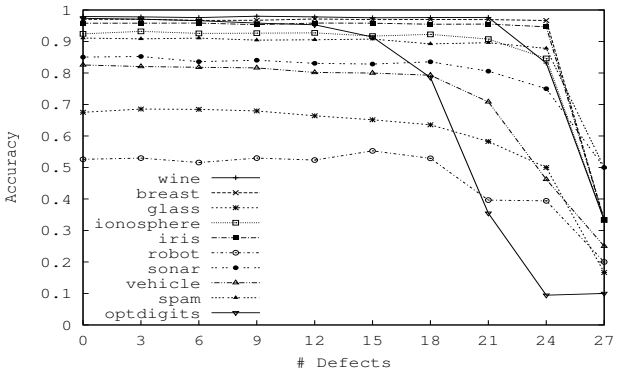


Fig. 10: *Accuracy vs. # of defects in input and hidden layers.*

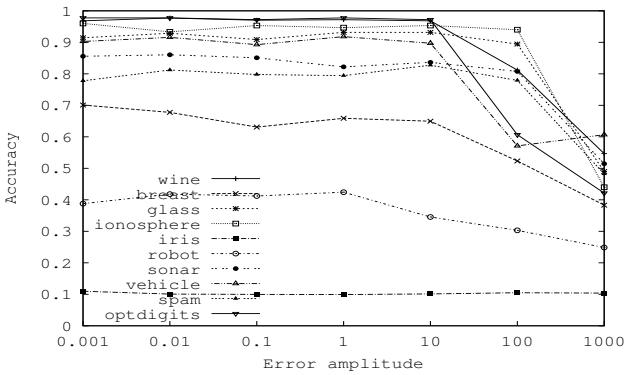


Fig. 11: *Accuracy vs. # error amplitude in the output layer.*

We reproduce the experiment 100 times, each time randomly varying the component where the defects occur; however the N defects of a network remain the same while the network is re-trained and tested on all target problems. For all applications considered, the accelerator can tolerate up to 12 defects, independently of which bit is affected, and most applications are not significantly affected by up to 20 defects. Even if errors occur on bits which significantly sway the neuron output, the neuron can be silenced by re-training. Afterward, accuracy can degrade sharply as too many neurons are affected.

In the output layer, a defect occurring at the activation function, or the adder just before the activation function, has a strong influence, though the rescaling effect of the activation function may tame it. In such cases, the network accuracy is fairly sensitive to the amplitude of the error introduced by the defect, as it directly affects the class value (or prediction). Within a digital neural network, the amplitude of the error depends on the position of the bit which incurred the error. We have introduced single defects in either of the two most sensitive parts of the output layer (adders, activation functions). For each such randomly generated faulty network, we evaluate the network accuracy after re-training; for each individual input row during the network test phase, we also measure the error amplitude at the adder output of the faulty neuron (the absolute value of the difference between the output of that adder with and without the defect). We then report the average accuracy as a function of the amplitude in Figure 11. For about half of the applications, such as `ionosphere`, the accuracy remains high as long as the error amplitude is not such that it can sway the class. However, for some applications, such as `iris` and `robot`, even tiny errors can sway the class and affect the accuracy.

Consequently, the adders and activation functions of the output layer should be considered a defect-sensitive part of the accelerator. For our network, these elements account for 25.9% of the output layer, and 2.3% of the total area. There are two possible ways to improve the defect tolerance of these elements of the network layer: consider the adders and activation functions of the output layer as key logic, or simply add spare (redundant) output neurons. The former solution is preferable as long as the fraction of the overall area covered by the output adders and activation functions is small; as technology scales down, the latter method will become more area efficient.

VII. Related Work

On hardware neural networks, there is a large body of work and it would not be possible to cite a representative subset here. Multiple implementations have been proposed, including commercial ones [15], such as the Intel ETANN [25], or recent ones for high-performance multimedia processing [33], but most are time-multiplexed. Some hardware networks are meant for industrial control [7], and others for smart sensor applications; many of the latter designs are low-power analog (or mixed-mode digital/analog) neural networks [48]. Analog logic is attractive because it is dense and low power, but the energy reductions and performance improvements are limited by their sensitivity to noise (temperature, mismatch,...). Moreover, noise accumulates over long chains of operators, limiting the number of bits carried per wire [9], or requiring to frequently convert into the digital domain for restoring the signal. Still, small-scale analog networks have been successfully shown to be applicable to high-performance systems, for instance for efficiently implementing a perceptron branch predictor [2]. Another popular field of research for analog hardware neural networks is biological emulation, with up to 60 million analog neurons implemented on one full wafer [46], but such implementations are only remotely related to computing applications.

While there is a large body of work on defect tolerance in hardware neural networks as well, there are few works simultaneously addressing high-performance, low-power and defect tolerance issues, especially at the transistor level. A not so recent but interesting example is the Philips LNeuro [40] which had to fulfill all these criteria due to a projected application in a satellite; though the design is time-multiplexed, it is modular for both scalability and reliability purposes. Most

of the research work on defect models for neural networks focus on the stuck-at-0 or stuck-at-1 defects in synaptic weights [20], [45], [22]. Special training algorithms have also been proposed to compensate for such stuck-at defects [26].

Beyond neural networks, there are several successful attempts at tackling transient faults or permanent defects within existing or modified processor architectures. Kumar et al. [30] show how a combined hardware and software approach can detect the occurrence of many transient faults. Kruijf et al. [29] introduce the notion that a degree of hardware inaccuracy can be tolerated provided the impact of errors on applications is controlled with algorithm-specific metrics. Chakrapani et al. [11] have even introduced transistor-level tradeoffs between energy and algorithm accuracy (PCMOS) and demonstrated applications to low-power ASICs; recently, Esmaeilzadeh et al. [19] have used similar concepts within processors themselves.

The approach used for determining the impact of hardware defects on the functional behavior of the neural network leverages existing test techniques [32], including expressing the impact of transistor-level defects, such as bridges [42], at the level of gates. However, unlike test techniques, the goal is not to find the appropriate test sequences which will allow to identify errors, but to attempt to reconstruct the output of the faulty circuit. Banerjee et al. [4] have shown that transistor-level defects cannot be modeled only using stuck-at faults at the input of logic gates, and can manifest themselves in many other ways (modification of the logic function, introduction of state, delays, etc). Alam et al. [1] recently outlined current trends in VLSI defects.

VIII. Conclusions and Future Work

Due to technology constraints, micro-architecture research may progressively shift its focus to accelerators, and especially energy-efficient and defect-tolerant accelerators. We make the case for hardware neural networks to be considered as one of the possible candidate accelerators. While hardware neural networks have already been considered in the past, we argue that the situation has drastically changed in terms of technology constraints, application scope, and even machine-learning algorithms. We outline the differences between past (time-multiplexed) designs and our proposed spatially expanded design, and especially that spatial expansion brings benefits in energy and defect tolerance. We particularly focus on validating and quantifying the intrinsic defect tolerance capabilities of hardware neural networks in the presence of transistor-level faults. The defect tolerance of neural networks proves to be not just a conceptual notion, but an actual property of hardware neural networks, provided the neural network is periodically retrained. As a result, it paves the way for hardware accelerators intrinsically capable of coping with hardware faults, without having to identify and disable the faulty parts.

We are currently working on two follow-up research paths. First, we want to illustrate the potentially broad application scope of a hardware neural network accelerator by demonstrating competitive implementations of multiple emerging high-performance tasks using neural network algorithms and run on a hardware neural network accelerator. Second, we want to increase the size of the neural networks that can be mapped without time-multiplexing, in order to efficiently tackle very large networks, such as Deep Networks. For that purpose, we are investigating three different approaches which could ultimately be combined: analog neurons for denser and more energy-efficient neurons, 3D stacking for the dense implementation of multiple network layers, and memristors for the dense implementation of synapses.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and advices.

References

[1] M. Alam, K. Roy, and C. Augustine. Reliability- and process-variation aware design of integrated circuits - a broader perspective. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 4A.1.1 –4A.1.11, april 2011.

[2] R. S. Amant, D. A. Jiménez, and D. Burger. Low-power, high-performance analog neural branch prediction. In *International Symposium on Microarchitecture*, pages 447–458, 2008.

[3] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[4] P. Banerjee and J. A. Abraham. Characterization and testing of physical failures in mos logic circuits. *IEEE Design and Test*, pages 76–86, Aug. 1984.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] S. Borkar. Design perspectives on 22nm cmos and beyond. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 93 –94, july 2009.

[7] B. K. Bose. Neural network applications in power electronics and motor drives – an introduction and perspective. *Industrial Electronics, IEEE Transactions on*, 54(1):14–33, Feb. 2007.

[8] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM.

[9] S. Chakrabartty and G. Cauwenberghs. Sub-microwatt analog vlsi support vector machine for pattern classification and sequence estimation. In *Advances in Neural Information Processing Systems (NIPS)*, Dec. 2004.

[10] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *International symposium on Computer Architecture*, page 247, Saint Malo, France, June 2010. ACM Press.

[11] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem. Probabilistic system-on-a-chip architectures. *ACM Transactions on Design Automation of Electronic Systems*, 12(3), August 2007.

[12] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *MICRO*, pages 225–236. IEEE, 2010.

[13] N. Clark, A. Hormati, and S. Mahlke. Veal. In *International Symposium on Computer Architecture*, pages 389–400, Beijing, June 2008.

[14] I. Corporation. Intel processor a100 and a110 on 90 nm process datasheet, January 2008.

[15] F. M. Dias, A. Antunes, and A. M. Mota. Artificial neural networks: a review of commercial hardware. *Engineering Applications of Artificial Intelligence*, 17(8):945 – 952, 2004.

[16] S. Draghici. On the capabilities of neural networks using limited precision weights. *Neural Netw.*, 15(3):395–414, 2002.

[17] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *TechnologyIntel Magazine*, 09, Feb 2005.

[18] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, June 2011.

[19] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In T. Harris and M. L. Scott, editors, *ASPLOS*, pages 301–312. ACM, 2012.

[20] J. Fieres, J. Schemmel, and K. Meier. A convolutional neural network tolerant of synaptic faults for low-power analog hardware. *Lecture Notes in Artificial Intelligence*, 4087:122–132, 2006.

[21] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *International Symposium on Computer Architecture*, page 37, New York, New York, USA, 2010. ACM Press.

[22] A. Hashmi, H. Berry, O. Temam, and M. H. Lipasti. Automatic abstraction and fault tolerance in cortical microachitectures. In *ISCA 2011, San Jose*, 2011.

[23] S. Haykin. *Neural Networks*. Prentice Hall Intl, London, UK, 2nd edition, 1999.

[24] J. Holi and J.-N. Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290, 1993.

[25] M. Holler, S. Tam, H. Castro, and R. Benson. *An electrically trainable artificial neural network (ETANN) with 10240 "Floating Gate" synapses*. IEEE Press, Piscataway, NJ, USA, 1990.

[26] T. Horita, T. Murata, and I. Takanami. A multiple-weight-and-neuron-fault tolerant digital multilayer neural network. In *Defect and Fault Tolerance in VLSI Systems, 2006. DFT '06. 21st IEEE International Symposium on*, pages 554–562, Oct. 2006.

[27] S. K. Jain and V. D. Agrawal. Modeling and Test Generation Algorithms for MOS Circuits. *IEEE Transactions on Computers*, 34(5):426–433, May 1985.

[28] D. Kegel. Crosstools, 2006.

[29] M. D. Kruijf, S. Nomura, and K. Sankaralingam. Relax : An Architectural Framework for Software Recovery of Hardware Faults. In *International Symposium on Computer Architecture*, Saint-Malo, 2010. ACM Press.

[30] S. Kumar, S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT : Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *International Symposium on Microarchitecture*, pages 122–132, New York, NY, 2009.

[31] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb. 2007.

[32] P. K. Lala. *An Introduction to Logic Circuit Testing*. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, 2008.

[33] D. Larkin, A. Kinane, and N. E. O'Connor. Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices. In I. King, J. Wang, L. Chan, and D. L. Wang, editors, *ICONIP (3)*, volume 4234 of *Lecture Notes in Computer Science*, pages 1178–1188. Springer, 2006.

[34] H. Larochelle, D. Erhan, A. C. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In Z. Ghahramani, editor, *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 473–480. ACM, 2007.

[35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[36] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *International Symposium on High Performance Computer Architecture*, page 105116, Raleigh, North Carolina, February 2009. IEEE Press.

[37] M.-l. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, 2008. ACM Press.

[38] G. M. Link and N. Vijaykrishnan. Thermal trends in emerging technologies. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 625–632, Washington, DC, USA, 2006. IEEE Computer Society.

[39] R. A. Maddox, G. Singh, and R. J. Safranek. *Weaving High Performance Multiprocessor Fabric Architectural Insights to the Intel QuickPath Interconnect*. Intel Press, July 2009.

[40] N. Mauduit, M. Duranton, J. Gobert, and J.-A. Sirat. Lneuro 1.0: a piece of hardware lego for building neural network systems. *Neural Networks, IEEE Transactions on*, 3(3):414–422, May 1992.

[41] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *International Symposium on Microarchitecture*, pages 210–222. IEEE Press, December 2007.

[42] W. Moore, C. Hora, M. Konijnenburg, and G. Gronthoud. A Gate-Level Method for Transistor-Level Bridging Fault Diagnosis. In *VLSI Test Symposium*, pages 266–271. IEEE Press, 2006.

[43] M. Muller. Dark Silicon and the Internet. In *EE Times "Designing with ARM" virtual conference*, 2010.

[44] S. Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Dpt. CS of Univ. of Copenhagen (DIKU), Oct 2003.

[45] D. Phatak and I. Koren. Complete and partial fault tolerance of feedforward neural nets. *Neural Networks, IEEE Transactions on*, 6(2):446–456, Mar 1995.

[46] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 431–438, June 2008.

[47] J.-s. Seo, B. Brezzo, Y. Liu, B. Parker, S. Esser, R. Montoye, B. Rajendran, J. Tierno, L. Chang, D. Modha, and Others. A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *IEEE Custom Integrated Circuits Conference*, pages 1–4. IEEE, Sept. 2011.

[48] F. Tenore, R. J. Vogelstein, R. Etienne-Cummings, G. Cauwenberghs, and P. Hasler. A floating-gate programmable array of silicon neurons for central pattern generating networks. In *International Symposium on Circuits and Systems (ISCAS 2006), 21-24 May 2006, Island of Kos, Greece*. IEEE, 2006.

[49] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. QsCORES : Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores Categories and Subject Descriptors. In *International Symposium on Microarchitecture*, 2011.